31

NOTE DI SOFTWARE

UNIVERSITA'
DEGLI STUDI DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e del Dipartimento di Scienze dell'Informazione dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie e bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione vengono revisionati dal Comitato di Redazione, che si avvale del contributo di specialisti del settore. Ogni contributo pubblicato riflette, comunque, le opinioni dei suoi autori, e non necessariamente quelle del Comitato di Redazione. Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
Viene distribuito ad aziende, Enti e specialisti del settore che ne facciano richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti - Stefania Bandini
Redazione: Franco Soresini

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FDPM-83 RNSW 120

31

FPDM - 83

NOTIZIARIO

Marzo 1986

Indice:

QUESTO NUMERO: Pag. 1

- UN AMBIENTE DI PROGRAMMAZIONE AUTORIFLESSIVO PER LINGUAGGI LOGICO-FUNZIONALI, di R. Ghislanzoni, L. Spampinato e G. Tornielli » 3
- UN'APPLICAZIONE DEL LINGUAGGIO PROLOG A PROBLEMI DI SPECIFICAZIONE DI SISTEMI NON SEQUENZIALI, di F. Furlan e G.A. Lanzarone » 15
- RAPPRESENTAZIONE LOGICA DI TESTI DI LEGGE, di R. Bertocchi e M. Stagnoli » 25
- U'INTERFACCIA UOMO-MACCHINA INTERATTIVA PER OBJ, di V. Besana e S. Zocco » 33
- SISTEMI ESPERTI E RAPPRESENTAZIONE ETEROGENEA DELLA CONOSCENZA IN MEDICINA: L'ESEMPIO DELL'EMERGENZA TOSSICOLOGICA, di S. Bandini e L. Spampinato » 41
- UN SISTEMA ESPERTO INTEGRATO CON MODULI DI GESTIONE E RAPPRESENTAZIONE DATI PER IL MONITORAGGIO POST-OPERATORIO DI SOGGETTI SOTTOPOSTI A BY-PASS DIGIUNO-ILEALE, di E. Lattuada, E. Lattuada, S.B. Doldi e P. Macchi » 51
- APPLICAZIONE DEI SISTEMI ESPERTI NELL'AUTOMAZIONE D'UFFICIO: IL PROTOTIPO CHAOS, di R. Vassallo e A. Zanaboni » 57
- SISTEMI DI INTERAZIONE DIDATTICA, di S.A. Cerri e P. Landini » 69

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

QUESTO NUMERO...

Questo numero di NOTE DI SOFTWARE raccoglie un insieme di contributi, provenienti da varie esperienze di ricerca del Dipartimento di Scienze dell'Informazione dell'Università di Milano, su temi tra loro diversificati, ma tutti riconducibili ad alcuni aspetti dell'Intelligenza Artificiale.

R. Ghislanzoni, L. Spampinato e G. Tornielli nel primo lavoro descrivono ALICE, un ambiente di programmazione per linguaggi logico-funzionali di tipo autoriflessivo, indirizzato allo sviluppo di applicazioni d'Intelligenza Artificiale.

Nello scritto seguente, invece, l'attenzione di F. Furlan e G.A. Lanzarone è rivolta ai problemi di specificazione di sistemi di telecomunicazione e all'applicazione del linguaggio Prolog in tale ambito.

Il linguaggio Prolog è tema d'interesse anche nel terzo lavoro, in cui R. Bertocchi e M. Spagnoli trattano della rappresentazione logica dei testi di legge.

V. Besana e S. Zocco descrivono poi un'interfaccia interattiva per la stesura di specifiche nel linguaggio OBJ, basata si di un editor diretto dalla sintassi.

I tre lavori successivi riguardano l'area dei sistemi esperti. S. Bandini e L. Spampinato illustrano alcuni problemi legati alla rappresentazione eterogenea della conoscenza mediante l'esempio di un sistema esperto per l'emergenza tossicologica.

Il lavoro di S.B. Doldi, E. Lattuada, E. Lattuada e P. Macchi riguarda invece un sistema esperto, costruito per mezzo di MICROEXPERT, integrato con moduli di gestione e di rappresentazione dati, per il monitoraggio di pazienti sottoposti a by-pass intestinale nella fase post-operatoria.

L'applicazione del sistema CHAOS nell'automazione d'ufficio è il tema del lavoro di R. Vassallo e A. Zanaboni; tale sistema è in grado di costruire una rappresentazione dell'organizzazione in essi inserito e di modificare la rappresentazione dell'organizzazione apprendendo dalle conversioni che gestisce.

Infine, S.A. Cerri e P. Landini presentano una rassegna volta ad illustrare alcune applicazioni realizzate dagli autori stessi nell'ambito ICAI (Intelligent Computer Aided Instruction).

La Redazione

UN AMBIENTE DI PROGRAMMAZIONE AUTORIFLESSIVO PER LINGUAGGI LOGICO-FUNZIONALI

Roberto Ghislanzoni, Giorgio Tornielli

Dipartimento di Scienze dell'Informazione - Università di Milano

Luca Spampinato - Quinary s.r.l. - Milano

RIASSUNTO

Il paradigma della riflessione procedurale proposto da Smith è di grande importanza nell'ambito dei linguaggi di programmazione autoreferenti: le possibilità di manipolazione esplicita dello stato della computazione offrono all'utente un ampio controllo per l'uso del sistema.

Lo schema canonico di Smith può essere utilizzato, modificando alcuni punti, per estendere la capacità riflessiva in ambiti differenti; uno degli scopi di questo lavoro consiste nel mostrare come sia possibile strutturare un interprete, con capacità riflessive, per clausole alla Horn.

Viene poi mostrato come, per mezzo della riflessione, venga resa possibile la comunicazione tra i due linguaggi che sono stati costruiti. Del sistema così creato vengono forniti alcuni brevi esempi d'uso.

ABSTRACT

Smith's theory about reflective languages is very important in A.I. field, particularly in languages with self-reference properties: the user can engender any kind of behaviour in his system by use of reflection.

Smith's ideas can be used in functional languages, as well as in logic programming: one of the main goals of this research is to build an interpreter for Horn clauses with the same reflective properties of Smith's interpreter. But the main goal of our work is to create a schemata, a general reflective architecture, to allow the communication between different languages by reflection: the system we built, called "Alice", can perform the communication between a user level and his description at meta-level. At meta-level you can see the interpreter below in two different kind of representation, i.e. functional and logic: the state of the computation is totally available to user. In "Alice", reflection represents a total change of your point-of-view: the reflection is now a full-dinamical-act: this is the great improvement with Smith's 3/Lisp.

We show you this new programming environment, and give a lot of example about reflection in *Alice*.

1. BASI CONCETTUALI DEL 3/LISP

L'interprete del 3/Lisp costruito da Smith basa la sua architettura sulla costruzione di una torre di interpreti (processori nella sua terminologia), con la caratteristica che un livello viene processato dall'interprete che si trova immediatamente sopra. La costruzione di tale torre è possibile modificando la visione che normalmente si ha di un processo computazionale seriale: Smith infatti propone di analizzare un processo dal punto di vista della *interpreting reduction* [6], secondo cui ogni processo non è un'entità atomica, ma ha al suo interno, come ora mostreremo, almeno un altro processo. Nel caso di un processo seriale, la prima grossolana suddivisione è fra strutture dati, la cui collezione è detta da Smith *campo strutturale,* e un processo in grado di operare su quelle strutture. Le strutture dati sono un'entità passiva, che contengono la descrizione del dominio oggetto, sia in termini di componenti fisiche, sia in termini di stati in cui gli elementi del dominio si possono trovare. Ovviamente non esiste limitazione su ciò che è esprimibile da una struttura dati: si può infatti esprimere conoscenza sia su un dominio concettuale (come fa un'asser-

zione del tipo "1 = succ(0)"), sia su un oggetto fisico: il campo strutturale in generale può essere pensato come il contenitore della conoscenza su cui opera un processo. La forma della conoscenza che viene espressa non è ovviamente rigida: infatti per un processo Lisp saranno utilizzate le cons-celle, mentre per un processo Apl un insieme di array.

Queste strutture dati non avrebbero però alcuna utilità, perchè puramente statiche. Deve così esistere un'entità, un processore, in grado di fornire la temporalità necessaria affinchè il processo possa effettivamente realizzare il comportamento che è contenuto staticamente nel campo strutturale sotto forma di dati e di programma.

Se il campo strutturale è statico, ciò non significa che debba necessariamente essere anche immutabile: difatti un processore regolarmente lo modifica durante una computazione (un assegnamento a variabili ne è un esempio).
Indubbiamente queste due componenti svolgono ruoli complementari e inscindibili all'interno di un processo computazionale: il processore fornisce la temporalità necessaria alle strutture per poter dire che il processo risultante ha un comportamento; mentre il campo strutturale contiene la conoscenza su cui il nuovo processo opera.

Dopo questa suddivisione, si è però ridotto un processo all'unione di un campo strutturale e di un ulteriore processo in grado di interpretare i dati che il campo contiene (fig. 1); la stessa riduzione è applicabile al nuovo processo interno: infatti esso non rappresenta null'altro che il comportamento del programma in grado di manipolare il campo strutturale; un programma è infatti definibile come manipolatore di strutture dati, quindi di elementi che appartengono al campo strutturale, piuttosto che al mondo esterno. Essendo un programma un'entità passiva, anch'esso rappresentabile come una collezione di strutture dati, è necessario che esista un ulteriore processo in grado di realizzare il comportamento che il programma contiene in forma statica (fig. 2).

A questo punto si potrebbe terminare la riduzione: infatti il nuovo processo ottenuto è l'interprete per il linguaggio; ma ci si può spingere ben oltre.
Infatti al processo creato può essere applicata un'ulteriore suddivisione: l'interprete può essere rappresentato come un programma scritto in uno specifico linguaggio, che nel caso del 3/Lisp viene scelto essere codificato mediante un PMC (Processor Meta Circolare), cioè un programma scritto in 3/Lisp medesimo, che mimi il comportamento dell'interprete stesso; operando in tal modo con successivi identici passi, si fa crescere all'infinito la torre (fig. 3)
Ogni livello che viene generato rappresenta l'interprete per il livello sottostante, di cui lo stato esplicito della computazione (cioè l'ambiente e la continuazione, nel caso del 3/Lisp, e l'ambiente e l'albero di ri-

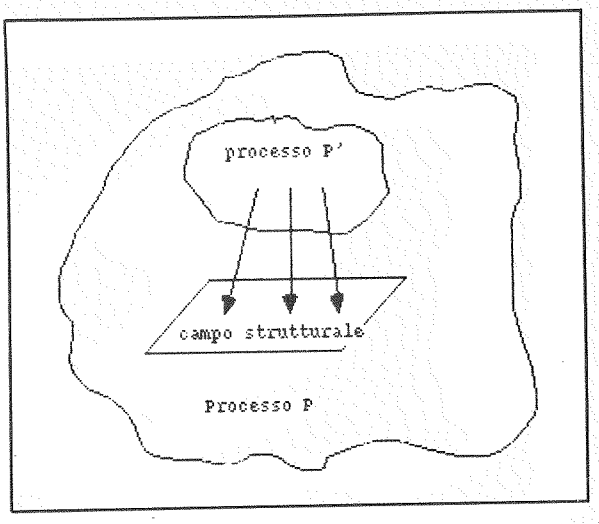


Fig. 1

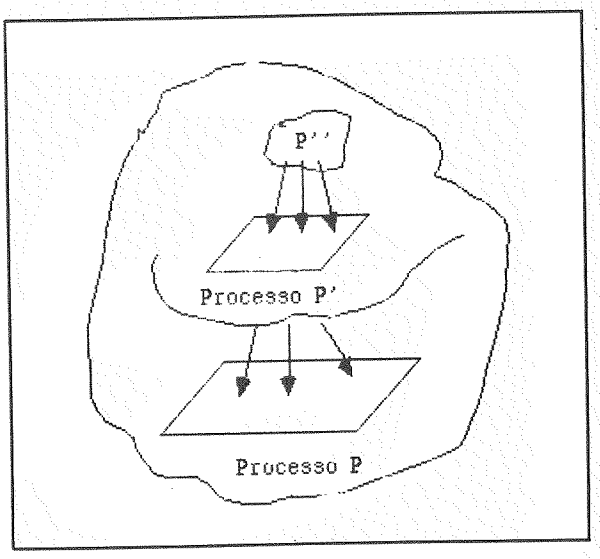


Fig. 2

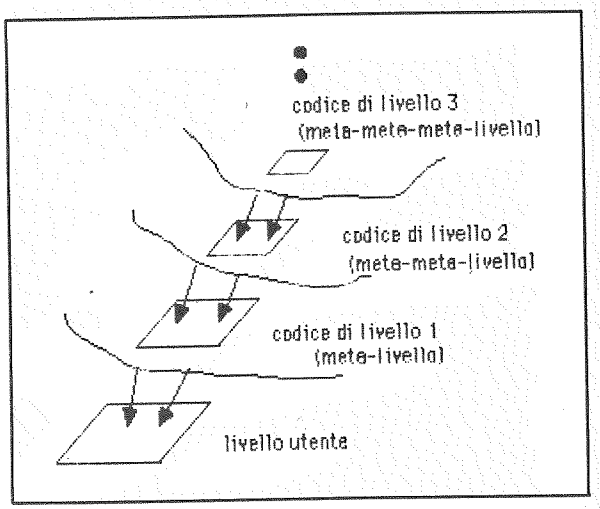


Fig. 3

cerca nel caso del 3/Logic) è appartenente al livello corrente, mentre i dati manipolati sono trattati per mezzo dei loro referenti.
In sostanza quindi le strutture dati che rappresentano il livello utente vedono sopra di loro un unico processo, che però è in realtà suddivisibile come mostrato in precedenza.

2. LA RIFLESSIONE PROCEDURALE

La costruzione della torre vista prima è fondamentale; per i nostri scopi, però ancora una cosa manca: si avverte infatti la necessità di avere un'importante caratteristica che permette di accedere allo stato del processore: denominiamo tale caratteristica *connessione causale*: sostanzialmente viene connotata in tre punti fondamentali:

1) Non si deve solo far riferimento al sistema che si intende descrivere, ma si deve disporre di una vera e propria teoria in modo tale da riuscire a giustificare ogni tipo di operazione che il sistema compie sugli elementi del linguaggio che può manipolare: tale operazione non deve essere parziale, ma definita su ogni elemento del sistema stesso. Nel 3/Lisp la teoria è rappresentata dal processore (PMC) che fa correre il livello attivato: tale processore è infatti in grado di descrivere pienamente il comportamento del programma utente. Ciò vale per ogni livello della torre.

2) Una volta fondata questa teoria, si deve essere in grado di metterla in relazione con il sistema che essa descrive: tali due entità devono quindi comunicare, e deve esistere la possibilità di rendere coscienti, l'uno dell'altro, il descritto e il descrittore: tale proprietà, che chiameremo *relazione causale,* deve instaurare un canale di comunicazione bilaterale tra gli eventi e le loro rispettive descrizioni. Questo significa che non solo è necessario poter accedere alla descrizione dello stato del sistema, sotto forma di struttura dati, ma soprattutto che ogni modifica compiuta su questa struttura dati deve influenzare il processo computazionale che da essa era descritto. Nel 3/Lisp questo viene ottenuto grazie alla riflessione procedurale.

3) Il terzo punto, forse meno concreto dei precedenti, sta nella considerazione che tale descrizione, o capacità di rappresentazione del sistema, non deve stare troppo lontano da esso, in modo da non perderlo di vista, ma non deve stargli nemmeno troppo vicino, correndo il rischio di confondersi con esso e di non definirsi con sufficiente autonomia: serve quello che Smith chiama "an appropriate viewpoint" del sistema che si intende descrivere.

Il primo passo per avere questa caratteristica cruciale è quello di poter disporre esplicitamente dello "stato" del processore: nella rappresentazione più sopra, lo stato corrente della computazione non è esplicito (si dice infatti che è "assorbito" nel processore); per renderlo visibile, il processore può, ad esempio, essere strutturato a continuazioni, in maniera tale che in ogni momento sia esprimibile lo stato della computazione.

Il punto fondamentale è che ciascun livello della torre fa correre il livello inferiore: dato che ogni livello è strutturato in maniera tale da aver rappresentate in termini espliciti le variabili che contengono la descrizione della computazione che si svolge sotto di lui, si può, con un atto di riflessione, accedere all'intera descrizione della computazione.
Si deve, però a questo punto, definire con maggiore chiarezza il concetto di riflessione che, per quanto sopra esposto, può essere visto sotto due prospettive differenti:

1) Esso rappresenta un "atto", grazie al quale si "sale" in maniera tangibile al livello del processore che stava interpretando (nella propria descrizione, quindi, e in modo totalmente dinamico).

2) Metaforicamente invita a considerare la torre prima descritta come una struttura simile ad un'architettura a "bambole cinesi", in cui tutti i processi stanno realmente correndo: nel punto 1) la visione non era così di ampio respiro; si presenta una situazione simile quando, ad esempio, si scrive un interprete per un linguaggio: richiamando l'eseguibile del sorgente, si ha un processo che lo esegue, ma si ha anche un processo di interpretazione per il linguaggio codificato a livello utente: entrambi i processi, per così dire, corrono l'uno dentro l'altro: sotto questo punto di vista non è questione di chiedersi quale dei livelli riflessivi sta attualmente girando, ma su quale di essi è focalizzata l'attenzione del momento.

Queste due interpretazioni, egualmente possibili, nascono dalla duplice natura dei PMC, visti, in un caso, dando maggiore peso alla loro natura di programmi, nell'altro, alla loro natura di processi.
Entrando ora nel dettaglio, una funzione riflessiva del 3/Lisp quindi è una funzione che non viene fatta correre allo stesso livello del codice che il processore sta interpretando, ma a livello del codice del processore medesimo, e dando quindi il controllo all'interprete che si trova *due* livelli sopra nella torre rispetto al codice utente (figura 4).
Una procedura riflessiva è specificata all'atto della dichiarazione, e tipicamente è della forma:

(lambda reflect [[arg1 arg2... argN] env cont] ‹body›)

"arg1", "arg2",... "argN" sono gli argomenti che riceve al momento della chiamata, ed "env" e "cont" rappresentano l'ambiente (environment) e la continuazione corrente del processore che stava interpre-

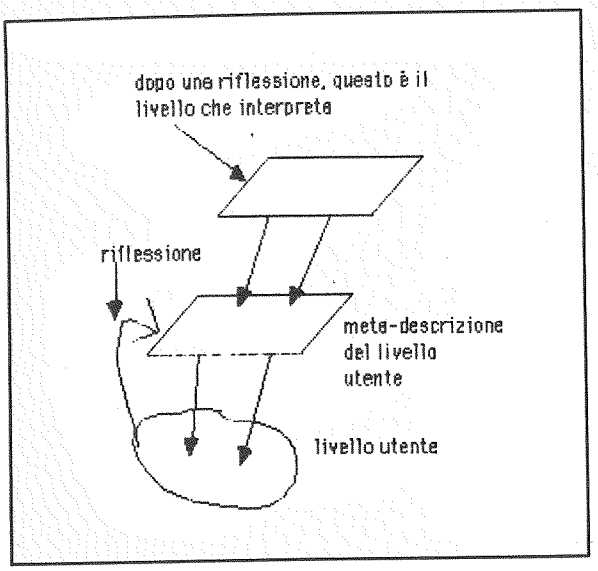


Fig. 4

tando: è la rappresentazione di uno stato che l'utente si trova totalmente a disposizione.
Vediamo ora qualche esempio di definizione e uso di procedure riflessive:

```
1> (define zero
       (lambda reflect [[] env cont]
           (cont '0)))
1='zero
```

Con tale funzione, quando utilizzata, viene specificato al processore, che in quel momento sta interpretando, di "mettere" tra il suo codice il codice della procedura riflessiva, e lì di farla "correre": si ricordi infatti che una procedura riflessiva viene interpretata dal processore soprastante quello corrente; la funzione appena descritta potrà, ad esempio, essere utilizzata nel seguente modo:

```
1> (+ 5 (zero))
1= 5
```

("1>" e "1=" sono i due prompt di *read* e di *reply* del 3/Lisp; il numero è quello del livello correntemente attivato).
Nel corpo della procedura riflessiva, la continuazione corrente (legata al parametro formale "cont") viene chiamata con il numerale "'0" come parametro (le continuazioni, se funzioni del processor come nel caso presente, maneggiano sempre i designatori degli argomenti): tale continuazione conteneva l'informazione di sommare qualcosa al numero 5: viene fornito questo qualcosa ("'0") e in questo modo al livello precedente (cioè "1") si trova il risultato sperato: grazie alla chiamatà di una funzione del processor, viene attivato esplicitamente un interprete: si scende al livello nella torre, quindi (questo *per ogni chiamata di funzione appartenente al processor*) e, dato che la continuazione conteneva l'informazione per reistanziare lo stato precedente, ci si trovava al livello da cui si era partiti prima di riflettere.

Un altro esempio riguarda la possibilità di definire delle funzioni di quotazione strutturale, come ad esempio la QUOTE, che non è definita nel linguaggio (non ve n'è infatti la necessità...):

```
1> (define quote
       (lambda reflect) [[arg] env cont]
           (cont 'arg)))
1= 'quote
1> (quote kiss)
1= 'kiss
```

Nella chiamata, l'argomento della funzione ("kiss"), che non viene normalizzato essendo la funzione riflessiva, viene ritornato alla continuazione alzato strutturalmente - quindi si ritorna il suo designatore: l'effetto è quello di ritrovare questo designatore al livello di partenza.
Per altri e più approfonditi esempi si vedano le pubblicazioni di Smith.

3. LA RIFLESSIONE COME ATTO DINAMICO

In Smith, il connotato di "reflect" associato ad una chiusura, non è per nulla influente sul significato della chiusura stessa: l'unica sua funzionalità è quella di ordinare all'interprete di considerarla in un certo modo differente dal suo comportamento standard: infatti una volta riconosciuta come riflessiva, una chiusura viene tranquillamente applicata dopo essere stata de-reflessa: allora, perchè non utilizzare la capacità riflessiva indipendentemente dall'oggetto chiusura, e considerarla un'entità a sè stante, in grado di porre qualsiasi cosa tra il codice dell'interprete che sta girando? Nella trasformazione specificata da una definizione di funzione non vi è comunque il connotato di riflessione, ed è quindi naturale tentare di fare dell'atto riflessivo un'entità in grado di avere vita autonoma all'interno del sistema. Inoltre, in 3/Lisp la riflessione è una proprietà statica, proprio perchè le funzioni riflessive sono dichiarate tali al momento della loro definizione; come conseguenza non si può definire una funzione applicabile sia al meta-livello sia al livello oggetto: anche quando le trasformazioni sono identiche bisogna definire due distinte chiusure.

Quest'ultimo punto è in contrasto con l'idea che il ragionamento di livello meta sia uguale a quello di livello oggetto [1]. Le funzioni esprimono solo delle regole di trasformazione tra un dominio oggetto o un dominio di meta-livello rispetto a quello corrente. Questa informazione non è catturata dal concetto di funzione ma rappresenta della conoscenza riguardo a quale dominio applicare suddetta funzione: quindi deve essere gestita dall'interprete al medesimo livello delle funzioni stesse.

Per questo si è introdotta una struttura dati con i propri costruttori e selettori che denota un atto di riflessione: il reflecter.
Limitandoci al Logic cioè al linguaggio basato sulle clausole di Horn con capacità riflessive analoghe a quelle del 3/Lisp, un reflecter contiene due informazioni: un simbolo di predicato che denota la relazione da calcolare al livello meta ed una sequenza di entità di livello oggetto che saranno viste anche al livello meta (naturalmente saranno viste come delle strutture dati poichè salendo nella torre aumentano anche i livelli di designazione).
Quando l'interprete incontra un reflecter, usa le informazioni in esso presenti per costruire la formula atomica da dimostrare al livello meta. In particolare sarà una formula che avrà il medesimo predicato contenuto nel reflecter, come primo termine la *rail* degli argomenti di livelli oggetto e come ultimi due termini, l'ambiente e l'albero della dimostrazione corrente.

Ad esempio se l'interprete durante una dimostrazione che ha come albero --tree-- e come ambiente --env--, trova il reflecter (il carattere "¯" rappresenta il tipo di dato riflessione):

¯ {A :x {B :y}}

costruisce la formula atomica

{A '[:x {B:y}] --env-- --tree--}

che poi dimostrerà al meta-livello.

Analogamente nella parte funzionale dell'interprete Lisp è possibile utilizzare questa capacità:

¯(f x y)

produrrà al meta-livello la struttura dati

(f '[x y] --env-- --cont--)

che sarà poi manipolata dall'interprete.

4. *ALICE,* UN'ARCHITETTURA RIFLESSIVA

Precedentemente sono stati presentati due linguaggi riflessivi, costruiti basandosi sulle idee originali di Smith: sorge ora spontanea una considerazione, originata dal fatto che due architetture assai simili sono a supporto dei linguaggi creati: avendo una genesi pressochè comune, non sarebbe possibile creare un unico linguaggio che sia la somma di tutti e due?

Si è convogliati quasi naturalmente verso questa via, stimolati anche dal ruolo che si sta configurando per la riflessione: sta assumendo via via una nuova connotazione in quanto si da maggior peso al suo uso a scapito della sua definizione: conseguentemente essa è diventata un'entità a sè stante all'interno del campo strutturale; nella visione che ora ci si apre, la riflessione potrebbe diventare l'"atto" grazie al quale realizzare la communicazione fra gli interpreti.

L'architettura realizzata è pertanto così fatta: per ogni livello attivano si hanno a disposizione due intepreti al meta-livello, uno funzionale e l'altro logico, che realizzano la descrizione (e la teoria) del suddetto livello. Entrambi hanno visione e accesso alla computazione corrente, che possono manipolare per influenzare il processo che stavano interpretando. Il modo per realizzare l'accesso allo stato della computazione è ancora una volta la riflessione procedurale: inoltre grazie ad essa si riesce a realizzare la comunicazione fra i due linguaggi che costituiscono *Alice:* infatti grazie all'atto riflessivo prima mostrato si riesce a specificare in quale descrizione si intende salire.
Una specificazione di riflessione funzionale permette di salire, dovunque si sia, verso l'interprete che realizza la descrizione funzionale del linguaggio che si sta utilizzando. E similmente per la specifica di una riflessione predicativa.

5. LOGIC

Il lavoro di ricerca è poi proseguito con la modellazione di un interprete per clausole di Horn. Lo schema architetturale che si è tenuto presente è quello del 3/Lisp di Smith, quindi anche per questo interprete valgono tutte le osservazioni fatte precedentemente: anche qui si ha alla base la torre di interpreti con capacità riflessive, ognuno realizzante la descrizione del linguaggio che sta utilizzando. Tale descrizione rappresenta inoltre la teoria del livello correntemente attivato: grazie ad essa si ottiene la connessione causale fra livello oggetto e meta-livello.

Con tale interprete si è cercato inoltre di mantenere la correlazione con quello che un dimostratore dovrebbe restituire: infatti nella manipolazione di una clausola la prima cosa che ci si chiede è se sia vera o meno all'interno della teoria corrente. L'interprete che abbiamo costruito rispetta pienamente questa esigenza, in quanto le uniche espressioni che può manipolare sono formule atomiche e congiunzioni, delle quali restituisce all'utente il valore di verità associato alla dimostrazione (ovviamente esistono anche tutti quei tipi di dati che sono necessari ad un interprete basato su clausole di Horn quali implicazioni e insiemi di clausole). Ad esempio la dimostrazione della formula atomica

{A :x}

avendo nella base dati {A 1}, restituisce all'utente il valore di verità "$t" pur istanziando la variabile "x" al valore "1". L'utente non può vedere i valori prodotti dell'unificatore in quanto l'interprete mantiene la correlazione con ciò che deve restituire. Lo stesso

avviene per la dimostrazione delle congiunzioni; infatti avendo nella base dati le clausole

⌊B 2⌋ ⌊C 3⌋

la dimostrazione della congiunzione

‹⌊B :X⌋ ⌊C :y⌋›

ritorna anch'essa il valore di verità "$t".

Chiaramente un interprete visto solamente con queste capacità può essere limitativo in quanto l'utente non ha mai visione dei valori prodotti all'unificatore, a differenza di quanto avviene negli interpreti Prolog in cui il risultato delle dimostrazioni è proprio l'insieme prodotto dall'unificazione.
Il modo di superare questo limite è quello di introdurre nel sistema la riflessione procedurale in modo tale che manipolando le strutture dati dell'interprete si riesca ad accedere allo stato della computazione ottenendo sia le informazioni richieste, sia la possibilità di modificare in maniera ancora più profonda il comportamento dell'interprete.

6. LA RIFLESSIONE DI *ALICE*

Si è visto come una delle caratteristiche principali di un sistema autoriflessivo sia di poter accedere, in qualsiasi istante, tramite riflessione, allo stato della computazione rappresentato in maniera esplicita: questa rappresentazione viene fatta in Smith per mezzo delle continuazioni, legate a lambda-espressioni: le ragioni per questa scelta di rappresentazione possono essere individuate in due punti:

1) Smith ha a che fare con un unico linguaggio: quindi anche al meta-livello, proprio perchè scritto nel medesimo linguaggio, è possibile manipolare al secondo ordine struttura dati legate a espressioni funzionali.

2) Nel momento in cui una continuazione passa da argomento di una funzione a chiamata esplicita, viene sfruttata la posizione in testa alla lista proprio perchè la variabile è legata ad una chiusura.

Nell'uso di un linguaggio come *Alice* sorgono invece problemi, in quanto al meta-livello non si ha un'unica descrizione del linguaggio che si sta usando, ma se ne hanno due.

Se si sta dimostrando, al meta-livello, come visto precedentemente, si ha una descrizione sia funzionale che predicativa dell'interprete Logic che si sta utilizzando, e ugualmente, se si sta normalizzando, la descrizione al meta-livello dell'interprete che sta correndo è costituita da due formalismi: si ha cioè un interprete per il Lisp scritto in Logic, e un interprete per il Lisp scritto in Lisp.

Considerando che lo stato della computazione deve essere esplicito e soprattutto manipolabile da entrambe le descrizioni, ciò porta alla conseguenza che, con le continuazioni espresse sotto forma di lambda-espressioni, si perderebbe capacità descrittiva, e ancor di più, connessione causale, con la meta-descrizione di colui che sta interpretando : e.g., se da un livello utente in cui è attivo un semplificatore, si sale nella descrizione Logic, si deve poter riconoscere e manipolare lo stato che ci si porta appresso: considerando che un interprete Logic non può indubbiamente manipolare delle lambda-espressioni, la descrizione dello stato della computazione deve essere fatta mediante una struttura dati di più basso livello rispetto al formalismo utilizzato da Smith.

Per quanto invece riguarda la salita da un livello utente logico alla corrispondente descrizione funzionale, non si ha alcun problema, perchè lo stato della computazione è interamente rappresentato dall'albero di dimostrazione, che è una struttura dati tranquillamente manipolabile da funzioni Lisp.
Stabilito quindi che la struttura dati rappresentante lo stato della computazione deve essere manipolabile da entrambi i formalismi, sorge il problema di come descrivere le continuazioni (useremo ancora questa terminologia per il seguito): si considerino i seguenti due casi (la funzione "normalize" è l'analoga della "eval" del Lisp tradizionale):

a) (normalize '1 global -??-)
b) ‹ ⌊global-env :env⌋ ⌊lnormalize '1 :env -??- l : res⌋›

Nel caso a) viene attivato un interprete Lisp: si opera una chiamata esplicita di una funzione apparentemente al processor, per cui si scende di livello nella torre istanziandovi un nuovo interprete e passandovi una continuazione definita dall'utente (indicata dal simbolo "-??-"): tale continuazione deve essere manipolabile da chi sta interpretando il nuovo processor istanziato, vale a dire dal medesimo livello utente da cui si era partiti.

Il caso b) presenta invece la chiamata di una funzione del processor ("lnormalize" è equivalente a "normalize": ne è la corrispondente descrizione predicativa) da parte di un dimostratore: il risultato della normalizzazione del primo argomento lo si ritroverà in ":res": il problema è che la continuazione passata deve essere manipolabile dal processor che sta interpretando, vale a dire da colui che attualmente è il livello utente: per conciliare questa esigenza, che sorge dalla diversa rappresentazione di strutture dati, si è pensato di introdurre per le continuazioni una struttura dati comune, di basso livello, che contenesse però tutte le informazioni necessarie: la sua definizione tipica è la seguente:

['atomo arg1 arg2... argN]

Per consentirne la manipolazione da parte di un in-

terprete funzionale, è necessario che nell'ambiente sia presente un lambda-legame per "atomo": all'interprete logic viene invece permessa la sua manipolazione grazie all'introduzione di un predicato che ha il compito di descrivere quale debba essere il comportamento dell'interprete ogni qualvolta incontri una sequenza siffatta: al fine di permettere il trattamento di questa particolare struttura dati, all'interprete sono stati aggiunti un predicato e una funzione che si occupano di questa computazione: sono "callcont" e "dimcont", rispettivamente per la parte funzionale e per la parte logica.

L'esempio precedente richiede quindi le seguenti definizioni (supponendo di volere fornire ad entrambi gli interpreti una continuazione che passi inalterato il risultato della normalizzazione):

```
'Lisp 1 › (define id
            (lambda simple [x] x))
'Lisp 1 = 'id
'Lisp 1 › (normalize '1 global ['id])
'Lisp 1 = '1

'Lisp 12 › (def-predicate 'dimcont
             '(dimcont ['id] :exp :exp))
'Lisp 12 = 'ok

'Horn 4 › ‹ (global-env :env) (normalize '1 :env
              ['id] :res)›
'Horn 4 = $t
```

I parametri di "dimcont" sono nell'ordine, la struttura dati rappresentante la continuazione, il valore che le viene passato dalla chiamata della funzione del processor, e la variabile ove "scaricare" il risultato: per la continuazione identica, il valore ricevuto deve essere passato tale e quale alla variabile.

7. ESEMPI D'USO DI *ALICE*

Grazie al fatto di avere a disposizione due diversi formalismi di rappresentazione, la possibilità di riflessione del sistema sono veramente degne di nota: in questo capitolo vengono mostrati svariati esempi delle funzionalità del sistema.

7.1 Il passaggio alla descrizione senza ritorno

In qualsiasi istante, dal livello correntemente attivo della torre ci si può trasferire a livello dell'interprete che sta attualmente girando: è infatti sufficiente scrivere un predicato o una funzione di quitting che non realizza la successiva discesa al livello di partenza: una semplice definizione di tale funzione può essere:

```
'Lisp 1› (define quit
           (lambda simple arg 'quitted))
'Lisp 1= 'quit
```

(si osservi il pattern-matching fatto con la sequenza degli argomenti, di cui, come vedremo ora, non ci importa nulla). Con questa funzione è possibile salire dal livello corrente, sia esso funzionale o logico, al livello della corrispondente descrizione funzionale.

```
'Lisp 1› ~(quit)
'Lisp 2= 'quitted                    ; -- oppure

'Horn 34› ~(quit)
'Lisp 35= 'quitted
```

La chiamata non ha argomenti, e le vengono passati in fase di salita della torre, al fine di mantenere la connnessione causale, l'environment e la continuazione correnti (o la teoria e l'albero, nel secondo esempio); il body della funzione si limita a restituire l'atomo "'quitted", dimenticandosi totalmente della descrizione del livello sottostante.
In maniera analoga si può definire un predicato di quitting per la salita alla descrizione Logic:

```
'Lisp 1› (def-predicate 'QUIT
           '(QUIT :args :env :cont))
'Lisp 1= 'QUIT
```

Il significato è molto chiaro: sale nella torre, dopo aver fatto pattern-matching con gli argomenti, portandosi dietro la continuazione e l'environment correnti, limitandosi poi a lasciare l'utente nel livello attivato.

7.2 Il passaggio alla descrizione con ritorno

Con le possibilità offerte dal sistema, si può voler salire, vedere lo stato della computazione corrente, e in seguito ridiscendere per continuare a fare ciò che si stava facendo prima: una semplice funzione che realizza questa performance, mostrando l'albero corrente di dimostrazione, è la seguente:

```
'Lisp 1› (define up&down
           (lambda simple [[] env tree]
             (block (output tree)
                    (newline)
                    (prove env tree))))
'Lisp 1= 'up&down
```

che può essere utilizzata nella dimostrazione della congiunzione corrente (avendo "(A 1) " nella base dati):

```
'Lisp 1› ~(QUIT)
'Horn 2= $t
'Horn 2› ‹ (A:x) ~(up&down) ›
  [ [ '‹(replay-logic 2 (global) '$t) ›
      [['(:x '1)]]
    [['(replay-logic 2 (global) '$t)
      '(clauses : reply-logic : compiled-pred)
      '‹›
    []]]]
'Horn 2= $t
```

La funzione ha rispettato il comportamento richiesto. Tutto ciò è pure possibile salendo nella descrizione Logic dell'interprete: vediamo invece il caso contrario, in cui, nel corso del programma di una computazione Lisp, si sale a livello della descrizione predicativa, si osserva quale sia lo stato della computazione, e si ridiscende per continuare ciò che si stava facendo in precedenza:

```
'Lisp 1› (def-predicate 'su&giu
          '(rule (su&giu :args :env :cont)
                 output :cont
                 (newline)
                 (dimcont :cont "upped :result)))
'Lisp 1= 'su&giu
'Lisp 1› (block (set x 100)
                ~(su&giu)
                (output 'done)
                (newline))
['block '[~(su&giu . []) (output 'done) (newline) ]
        (global) ['replycont 1 (global) ] ]
'done
'Lisp 1= 'ok
```

Si nota come sia totalmente esplicito lo stato della computazione sottostante, rappresentato dalla struttura dati che contiene le informazioni sulla continuazione della block.
Con l'esempio mostrato può essere evidenziata la connessione di causa tra il livello utente e la sua descrizione: si modifichi infatti il codice della "su&giu" come segue:

```
'Lisp 1› (def-predicate 'su&giu
           '(rule (su&giu :args :env :cont)
                  (last :cont :lastcont)
                  (output :lastcont)
                  (newline)
                  (dimcont :lastcont "upped :result )))
'Lisp 1= 'su&giu
'Lisp 1› (block (set x 100)
                ~(su&giu)
                (output 'done)
                (newline))
['replycont 1 (global) ]
'Lisp 1= 'upped
```

Si nota ora come, avendo tolto una parte della continuazione, più precisamente quella che regolava il proseguimento della "block", si siano avute pesanti ripercussioni sulla computazione sottostante: infatti l'atomo "'upped" viene ora passato alla "replycont", e non più perso, come avveniva in precedenza ed inoltre la stampa di "done" non viene fatta.
A titolo di ulteriore esempio, viene qui mostrata la definizione, in due differenti formalismi, della funzione di quotazione strutturale "quote":

```
'Lisp 1› (define quote
           (lambda simple [[arg] env cont]
                  (callcont cont ^arg env)))
'Lisp 1= 'quote
'Lisp 1› (def-predicate 'QUOTE
           '(rule (QUOTE :args :env :cont)
                  (1st :args :exp)
                  (up :exp up-exp)
                  (dimcont :cont :up-exp :result)))

'Lisp 1= 'QUOTE

'Lisp 1›  (quote a)
'Lisp 1= 'a
'Lisp 1› ~(QUOTE b)
'Lisp 1= 'b
```

7.3 Le definizioni di alcuni predicati dei linguaggi logici

In *Alice* è possibile definire, grazie alla riflessione, alcuni dei predicati che normalmente si incontrano in un ambiente di programmazione logica:

• *Il predicato fail*

È il predicato mai dimostrabile, che porta al fallimento della dimostrazione della frontiera dell'albero: si può definire come una funzione Lisp invocabile da Logic:

```
'Lisp 1› (define fail
           (lambda simple [[] env tree]
                  (prove env (clear-and-level tree)))
'Lisp 1= 'fail
```

L'effetto è quello di salire nella descrizione funzionale dell'interprete e di togliere il livello *and* dell'albero attuale di dimostrazione: alla successiva chiamata di "prove" si riprenderà a dimostrare dal prossimo livello *or* (se presente), ritornando al livello dell'interprete prima lasciato.

• *Il predicato true*

È il predicato sempre dimostrabile: è estremamente semplice da definire, in quanto è sufficiente la seguente:

```
'Lisp 1› (define true
           (lambda simple [[] env tree]
                  (prove env tree))
'Lisp 1= 'true
```

La chiamata "~(true) risulterà sempre dimostrabile.

• *Il predicato print*

Di norma dopo una dimostrazione viene restituito il valore di verità $t o $f, ad indicare se la richiesta era provabile o meno; per vedere i valori di unificazione delle variabili si può ricorrere alla seguente funzione:

```
'Lisp 1› (define print
           (lambda simple [[var] env tree]
             (block (output
                      (expand
                        var
                        (unif-env-and
                          (and-level tree))))
                    (newline)
                    (prove env tree))))

'Lisp 1= 'print
```

può ad esempio, essere utilizzata nel modo seguente:

```
'Horn 1› ‹ |A :x| ~(print :x)›
'1
'Horn 1= $t
'Horn 1›
```

• *Il predicato assert*

È il classico predicato Prolog che permette di aggiungere asserzioni all'interno della base dati: in *Alice* è possibile scrivere una funzione che porta a termine la identica operazione:

```
'Lisp 1› (define assert
           (lambda simple [args env tree]
             (let [[clauses (cscons-rail args)]]
               (block
                 (rebind
                   (csname clauses)
                   (mergecs
                     (binding-logic
                       (csname clauses)
                       env)
                     clauses)
                   (env)
                 (prove env tree)))))

'Lisp 1= 'assert

'Horn 12› ~(assert |A 1| |A 2|)
'Horn 12= $t

'Lisp 2› A
'Lisp 2= |clauses: |A 1| |A 2||
```

La funzione, molto semplicemente, si limita a legare nell'ambiente, al nome di predicato delle clausole, l'unione ("mergecs") del legame precedente con ciò che viene passato dall'utente; si chiama poi la dimostrazione a cui si era avuto accesso.

•• **ask**

In alcune applicazioni non è sufficiente considerare l'insieme delle definzioni note all'interprete come l'intera base dati su cui il programma opera, ma si può anche considerare l'utente stesso come una sua estensione. In questo modo, quando una formula non è dimostrabile nelle basi dati dell'interprete, è necessario che sia l'utente a deciderne la dimostrabilità. La meta-funzione **ask** ha proprio questo scopo. La sua definizione è la seguente:

```
(lambda simple [[to-prove] env tree]
  (let [[unif (unif-env-and (and-level-tree))]]
    (cond [(prove env (make-tree to-prove unif))
           (prove env tree)]
          [$t
           (block
             (output 'dimostrabile?)
             (output (expand to-prove unif))
             (newline)
             (if (= (input) 'yes)
             (prove env tree)
             (prove env (clear-and-level tree))))]])))
```

la meta-funzione ask può essere usata nel seguente modo:

```
'Horn 1› |A 1|
'Horn 1= $f
'Horn 1› ~(ask |A 1|)
'dimostrabile? '|A 1|
'yes
'Horn 1= $t
```

•• **my-if**

Analogicamente la capacità di riflessione può essere sfruttata per scrivere in Logic le strutture di controllo per il 3/Lisp. Ad esempio mostriamo come sia possibile ridefinire l'**if** del 3/Lisp in Logic. La sua definizione è la seguente:

```
|clauses:
    |rule
      |my-if :args :env :cont|
     |1st :args :test|
        |normalise :test :env ['id] '$t|
        |2nd :args :then|
        |normalise :then :env :cont :result|
    |rule
      |my-if :args :env :cont|
     |1st :args :test|
        |normalise :test :env ['id] '$f|
        |3rd :args :else|
        |normalise :else :env :cont :result|
```

Questo predicato può essere utilizzato come la comune **if**:

```
'Lisp 1› ~|my-if (=1 1)
            (block
              (output 'yes)
              (newline))
            (block
```

```
            (output 'no)
            (newline))
'yes
'Lisp 1= 'ok
```

8. RINGRAZIAMENTI

Si ringrazia il prof. G. Degli Antoni, per il costante interessamento e i continui spunti offerti al lavoro.

9. BIBLIOGRAFIA

[1] Batali, J. COMPUTATIONAL INTROSPECTION, M.I.T. Artificial Intelligence Laboratory Memo AIM 701, 1983

[2] Des Rivières, J. and Smith, B.C. THE IMPLEMENTATION OF PROCEDURALLY REFLECTIVE LANGUAGES, Palo Alto, Xerox PARC ISL-4, 1984

[3] Ghislanzoni, R., UN AMBIENTE DI PROGRAMMAZIONE AUTORIFLESSIVO PER LINGUAGGI LOGICO-FUNZIONALI. ASPETTI ARCHITETTURALI CON PARTICOLARE RIFERIMENTO AI LINGUAGGI FUNZIONALI, Tesi di laurea del corso di Scienze dell'Informazione, a.a. 1984-1985

[4] Ghislanzoni, R., Samek Ludovici, V. Tornielli, G., H: UN LINGUAGGIO DI PROGRAMMAZIONE LOGICA, Note di Software, 27-28, 1985, pgg. 23-31

[5] Smith, B.C. des Rivières, J. INTERIM 3-LISP REFERENCE MANUAL, Palo Alto, Xerox PARC ISL-1, Technical Report, 1984

[6] Smith, B.C. THE IMPLEMENTATION OF PROCEDURALLY REFLECTIVE LANGUAGES, Palo Alto, Xerox PARC ISL-4, 1984

[7] Smith, B.C. REFLECTION AND SEMANTICS IN LISP, Palo Alto, Xerox PARC ISL-5, 1984

[8] Smith, B.C. THE COMPUTATIONAL METAPHOR, disponibile dall'autore, 1982

[9] Tornielli, G. UN AMBIENTE DI PROGRAMMAZIONE AUTORIFLESSIVO PER LINGUAGGI LOGICO-FUNZIONALI. ASPETTI ARCHITETTURALI CON PARTICOLARE RIFERIMENTO AI LINGUAGGI LOGICI, Tesi di laurea del corso di Scienze dell'Informazione, a.a. 1984-1985

APPENDICE

L'interprete Lisp per il Lisp

```
(define read-normalise-print
  (lambda simple [level env]
    (normalise (prompt&read 'Lisp level) env
      ['replycont level env])))


(define replycont
  (lambda simple [result level env]
    (block
      (prompt&reply 'Lisp result level)
      (read-normalise-print level env))))


(define normalise
  (lambda simple [exp env cont]
    (cond [ (normal exp)
            (callcont cont exp env) ]
          [ (atom exp)
            (callcont (binding exp env) env) ]
          [(rail exp)
           (normalise-rail exp env cont) ]
          [ (reflecter exp)
            (normalise (mname exp) env
              ['red-ref (mterms exp) env cont)]]
          [ (pair exp)
            (reduce (car exp) (cdr exp) env cont)])))


(define red-ref
  (lambda simple [name! terms env cont]
    (\name! terms env cont)))


(define reduce
  (lambda simple [proc args env cont]


(define proccont
  (lambda simple [proc! args env cont]
    (normalise args env
      ['argscont proc! env cont])))

(define argscont


(lambda simple [args! proc! env cont]
  (if (primitive proc!)
      (callcont cont ^(\proc! . \args!) env)
      (normalise
          (body proc!)
          (bind (pattern proc!)
                args!
                (environment proc!))
          cont))))


(define normalise-rail
  (lambda simple [rail env cont]
    (if (empty rail)
        (callcont cont (rcons) env)
        (normalise (1st rail) env
           ['firstcont rail aenv cont]))))

(define firstcont
  (lambda simple [first! rail  env cont]
    (normalise-rail (rest rail) env
      ['restcont  first! env cont])))


(define restcont
  (lambda simple [rest! first! env cont]
    (callcont cont (prep first! rest!) env)))


(define callcont
  (lambda simple [cont exp env]
    (\ (binding (1st cont) env) . (prep exp (rest cont))))))
```

L'interprete Lisp per Logic

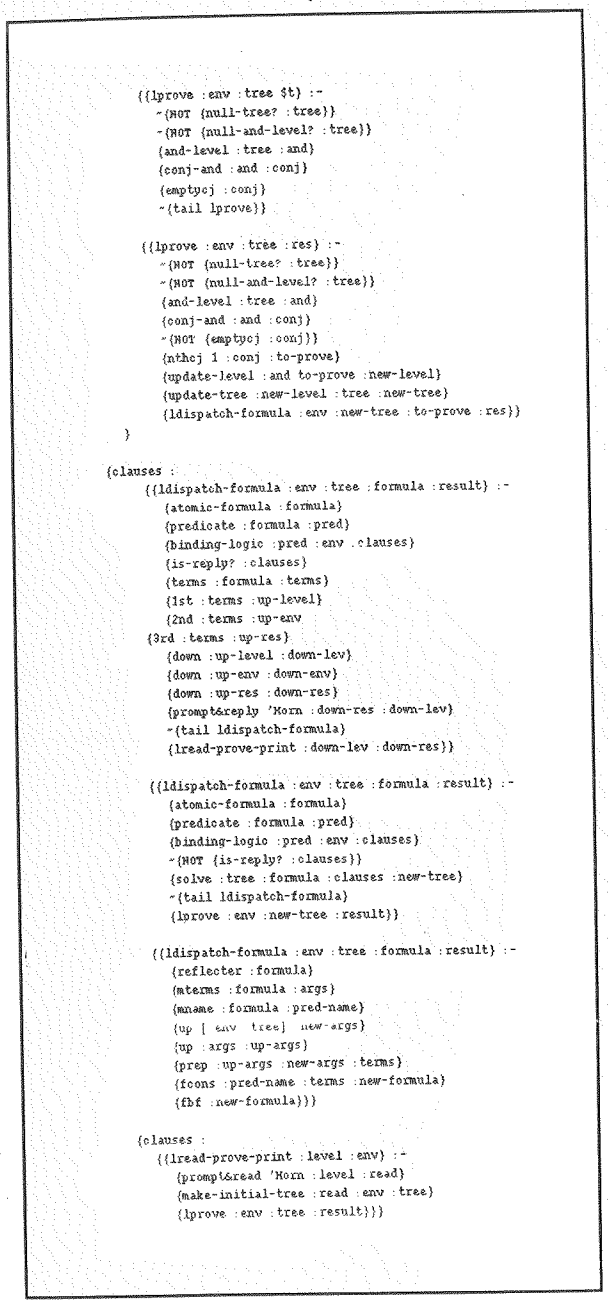
```
(define prove
  (lambda simple [theory tree]
    (cond [ (null-tree? tree) $f)
          [ (null-and-level? tree)
            (let [[next-or (1st (or-levels tree))]]
              (prove theory
                     (solve (new-tree tree next-or)
                            (formula-or next-or)
                            (clauses-or next-or))))]
          [$t
            (let [[next-level (and-level tree)]]
              (if (emptyoj (conj next-level))
                  $t
                  (let [[to-prove (nthoj 1 (conj next-level))]]
                    (dispatch-formula
                           theory
                           (update-tree
                              (update-level  next-level to-prove)
                              tree)
                           to-prove))))])))


(define dispatch-formula
  (lambda simple [theory tree formula]
    (let [[clauses
             (binding-logic (predicate formula) theory)]]
      (cond [ (is-reply? clauses)
              (let [[ter  (terms formula)]]
                (block
                    (prompt&reply 'Horn \(3rd ter)  \(1st ter))
                    (read-prove-print \(1st ter) \(2nd ter))))]
            [ (atomic-formula formula)
              (prove theory
                     (solve tree formula clauses))]
            [ (reflecter formula)
              (normalise (mname formula) theory
                 ['red-ref (mterms formula) theory tree])])))))
```

L'interprete Logic per il Lisp

```
{clauses:
  {rule
    {lread-normalise-print :level :env}
      {prompt&reply 'Lisp :level :read}
      {lnormalise :read :env
        ['replycont :level :env ] :result}}}

{clauses:
  {rule
     {lnormalise :exp :env :cont :result}
       {normal :exp}
       ~{tail lnormalise}
       {dimcont :cont :exp  :result}}
  {rule
     {lnormalise :exp :env :cont :result}
       {atom :exp}
       {binding :exp :env :value}
       ~{tail lnormalise}
       {dimcont :cont :value :result}}
  {rule
     {lnormalise :exp :env :cont :result}
       {rail :exp}
       ~{tail lnormalise}
       {lnormalise-rail :exp :env :cont :result}}
  {rule
     {lnormalise :exp :env :cont :result}
       {reflecter :exp}
       {mname  :exp :name}
       {mterms  :exp :temas}
       {up :terms :upped-terms}
       {up [ :env  :cont] :upped-state]
       {prep :upped-terms:upped-state :new-terms}
       {fcons :name :new-terms :formula}
       ~{tail lnormalise}
       {fbf :formula}}
  {rule
     {lnormalise :exp :env :cont :result}
       {pair :exp}
       {car :exp :proc}
       {cdr :exp :args}
       {lreduce :proc :args :env :cont :result}}}

{clauses:
  {rule
     {lreduce :proc :args :env :cont :result}
       {lnormalise :proc :env
         ['proccont :args :env :cont] :result}}}

{clauses:
  {rule
     {lnormalise-rail :rail :env :cont :result}
       {empty :rail}
       ~{tail lnormalise-rail}
       {dimcont :cont '[] :result}}
  {rule
     {lnormalise-rail :rail :env :cont :result}
       ~{NOT (empty :rail)}
        {1st :rail :first}
       ~{tail lnormalise-rail}
        {lnormalise :first :env
            ['firstcont :rail :env :cont] :result}}}

{clauses:
  {rule
     {dimcont ['replycont :level :env] :exp :result}
       {prompt&reply 'Lisp :exp :level}
       ~{tail dimcont}
       {lread-normalise-print :level :env}}
  {rule
     {dimcont ['proccont :args :env :cont] :proc! :result}
       ~{tail dimcont}
       {lnormalise :args :env
         ['argscont :proc! :env :cont] :result}}
  {rule
     {dimcont ['argscont :proc! :env :cont] :args! :result}
       {primitive :proc!}
       {apply-primitive :proc! :args! :value }
       ~{tail dimcont}
       {dimcont :cont :value :result}}

  {rule
     {dimcont ['argscont :proc! :env :cont] :args! :result}
       ~{NOT {primitive :proc!}}
       {body :proc! :body}
       {environment :proc! :static-env}
       {pattern :proc! :pattern}
       {bind :pattern :args! :static-env :new-env}
       ~{tail dimcont}
       {lnormalise :body :new-env :cont :result}}
  {rule
     {dimcont ['firstcont :rail :env :cont] :first! :result}
       {rest :rail :rest}
       ~{tail dimcont}
       {lnormalise-rail :rest : env
         ['restcont :first!  :env :cont] :result}}
  {rule
     {dimcont  ['restcont :first! :env :cont] :rest! :result}
       {prep :first! :rest! :new-rail}
       ~{tail dimcont}
       {dimcont :cont :new-rail :result}}}
```

L'interprete Logic per Logic

```
{ clauses :
    {{lprove :env :tree $f} :-
       {null-tree? :tree}
       ~{tail lprove}}

    {{lprove :env :tree :res} :-
       ~{NOT {null-tree? :tree}}
       {null-and-level? :tree}
       {or-levels :tree :ors}
       {1st :ors :next-or}
       {new-tree :tree :next-or :new-tree}
       {formula-or :next-or :formula}
       {clauses-or :next-or :clauses}
       {solve :new-tree :formula :clauses :solved-tree}
       ~{tail lprove}
       {lprove :env :solved-tree :res}}
```



```
    ((lprove :env :tree $t) :-
      ~(HOT (null-tree? :tree))
      ~(HOT (null-and-level? :tree))
      (and-level :tree :and)
      (conj-and :and :conj)
      (emptycj :conj)
      ~(tail lprove))

    ((lprove :env :tree :res) :-
      ~(HOT (null-tree? :tree))
      ~(HOT (null-and-level? :tree))
      (and-level :tree :and)
      (conj-and :and :conj)
      ~(HOT (emptycj :conj))
      (nthcj 1 :conj :to-prove)
      (update-level :and to-prove :new-level)
      (update-tree :new-level :tree :new-tree)
      (ldispatch-formula :env :new-tree :to-prove :res))
  )

(clauses :
    ((ldispatch-formula :env :tree :formula :result) :-
      (atomic-formula :formula)
      (predicate :formula :pred)
      (binding-logic :pred :env :clauses)
      (is-reply? :clauses)
      (terms :formula :terms)
      (1st :terms :up-level)
      (2nd :terms :up-env
     (3rd :terms :up-res)
      (down :up-level :down-lev)
      (down :up-env :down-env)
      (down :up-res :down-res)
      (prompt&reply 'Horn :down-res :down-lev)
      ~(tail ldispatch-formula)
      (lread-prove-print :down-lev :down-res))

     ((ldispatch-formula :env :tree :formula :result) :-
      (atomic-formula :formula)
      (predicate :formula :pred)
      (binding-logic :pred :env :clauses)
      ~(HOT (is-reply? :clauses))
      (solve :tree :formula :clauses :new-tree)
      ~(tail ldispatch-formula)
      (lprove :env :new-tree :result))

     ((ldispatch-formula :env :tree :formula :result) :-
      (reflecter :formula)
      (mterms :formula :args)
      (mname :formula :pred-name)
      (up [ env  tree]  new-args)
      (up :args :up-args)
      (prep :up-args :new-args :terms)
      (fcons :pred-name :terms :new-formula)
      (fbf :new-formula)))

    (clauses :
      ((lread-prove-print :level :env) :-
       (prompt&read 'Horn :level :read)
       (make-initial-tree :read :env :tree)
       (lprove :env :tree :result)))
```

UN'APPLICAZIONE DEL LINGUAGGIO PROLOG A PROBLEMI DI SPECIFICAZIONE DI SISTEMI NON SEQUENZIALI

F. Furlan, G.A. Lanzarone
Dipartimento di Scienze dell'Informazione - Università di Milano

RIASSUNTO

Si descrivono problemi di specificazione di sistemi di telecomunicazioni, e di verifica delle specifiche scritte nel linguaggio SDL, standard in questo settore. Per verificare la corretta comunicazione tra i processi specificati in SDL, si passa a una rappresentazione in reti di Petri. Poichè per sistemi reali le reti risultano di grandi dimensioni, si introduce un metodo di riduzione che ne conserva le proprietà da verificare. Si descrivono le principali caratteristiche dell'implementazione in Prolog di uno strumento che realizza questo metodo. Esso consiste in un programma di traduzione da SDL a reti di Petri, di un programma di riduzione di reti, e di un programma di analisi delle reti ridotte. Si discutono i vantaggi di compattezza, uniformità e prototipazione veloce che l'utilizzo di Prolog consente nel realizzare queste diverse componenti dello strumento.

ABSTRACT

Specification problems of telecommunication systems, and problems of verification of specifications written in SDL, the standard language in this field, are described. In order to verify the correct communication among processes specified in SDL, a representation with Petri nets is derived.
Since specifications of real systems result in large nets, a reduction method is introduced, which preserves the properties to be verified. The main features of a Prolog implementation of this method are described. The implemented tool consists of a program to translate from SDL to Petri nets, a program to reduce the nets, and a program to analyze the reduced nets.
Advantages of compactness, uniformity and rapid prototyping, obtained by using Prolog to implement such different components of the tool, are discussed.

1. DEFINIZIONE DEL PROBLEMA

Per la specificazione dei sistemi di telecomunicazioni, il CCITT (Comité Consultatif International Telegraphique et Telefonique) ha definito il linguaggio SDL (Specification and Description Language), proponendolo come standard internazionale per costruttori (società manifatturiere) ed acquirenti (società di esercizio, amministrazioni PPTT) di tali sistemi [3].

Dopo diversi anni dalla sua iniziale definizione, SDL è un linguaggio ampiamente utilizzato nelle applicazioni del settore nei vari paesi, e allo stesso tempo in ulteriore evoluzione. Probabilmente nessun altro ampio settore applicativo (soprattutto nel campo dei sistemi non sequenziali) gode nella stessa misura dell'affermazione e dell'uso effettivo di un unico linguaggio di specificazione (con i vantaggi e con i limiti che si discuteranno nel seguito).

L'utilizzo di SDL incontra uno dei classici requisiti dell'ingegneria del software, quello di specificare un sistema prima di implementarlo. Un più recente requisito richiede che le specifiche siano anche verificabili e, possibilmente, eseguibili. Lo scopo è di estendere il più possibile l'individuazione di errori o comunque di possibili fonti di problemi nel progetto di un sistema, prima della sua implementazione, per diminuirne i costi di sviluppo e aumentarne l'affidabilità.

SDL non incontra questo secondo requisito, in quanto non contiene esplicitamente astrazioni utili alla ve-

rificabilità ed eseguibilità. Gli utilizzatori di SDL lo hanno perciò finora impiegato a scopi principalmente descrittivi, con il beneficio di un unico linguaggio di specificazione noto internazionalmente nel settore, ma senza il supporto di metodi, e conseguenti strumenti, di analisi e verifica delle specifiche.

In ambito CCITT sono allo studio possibili estensioni di SDL che portino a soddisfare questo secondo requisito. In questo articolo si illustreranno i risultati di alcuni lavori compiuti in questa direzione.

2. CARATTERISTICHE DI SDL

SDL (per un'introduzione al linguaggio si veda [20]) è basato sul paradigma di strutturazione di un sistema non sequenziale in un insieme di processi sequenziali comunicanti a scambio di messaggi. Il numero dei processi è definito staticamente; ogni processo è rappresentato come una macchina a stati finiti estesa, consistente di un insieme di stati, di transizioni, di input, di output, di operazioni e di decisioni.

La comunicazione tra i processi è asincrona, mediante code di messaggi, una per ogni processo: un processo può comunicare con gli altri ponendo messaggi nella rispettiva coda, e viceversa ricevendo, dagli altri, messaggi nella propria coda, da cui preleva per l'elaborazione durante la propria evoluzione. Le code seguono una disciplina FIFO in ordine di arrivo (ad eccezione del costrutto SAVE che si vedrà più avanti). La comunicazione a scambio di messaggi (segnali semplici o parametrici) è la sola fonte di conoscenza reciproca tra i processi, che non dispongono di una memoria condivisa.

La specifica di un processo descrive, per ogni suo stato, un insieme di possibili messaggi attesi in input, e per ognuno di essi la transizione da compiere fino a uno stato successivo; la transizione può contenere l'output di messaggi verso altri processi, la modifica di valori di variabili proprie del processo e la valutazione di decisioni su tali variabili private, il cui esito determina la scelta dello stato successivo.

La semantica, operazionale e informale, di tale specifica è la seguente. A partire da uno stato designato come iniziale, e in ogni suo altro stato, il processo è in attesa di un input, per il quale esamina la sua coda:

- se la coda è vuota, il processo rimane in attesa in quello stato;
- se la coda non è vuota, esamina il primo messaggio:
  - se è uno degli input possibili per quello stato, lo consuma eseguendo la transizione specificata sino al corrispondente stato successivo;
  - se è un input diverso da quelli possibili, lo consuma senza eseguire alcuna transizione, ed esamina il successivo;
  - se è un input specificato come "SAVE" per quello stato, lo lascia nella coda ed esamina il successivo.

L'evoluzione di un processo si attua quindi con transizioni da uno stato all'altro, durante le quali avvengono comunicazioni in entrambe le direzioni con altri processi, fino a raggiungere nuovamente lo stato iniziale: i processi sono ciclici.

Tra i vantaggi del modello SDL vi è quello di consentire la decomposizione di un sistema in processi sequenziali comunicanti, che si ritiene un buon criterio di ristrutturazione (cfr. ad es. [10], [15], [18]). Ne risulta una specifica indipendente dall'architettura fisica del sistema da sviluppare (mono o multi-processor), non molto impegnativa rispetto all'implementazione, e allo stesso tempo abbastanza concreta da essere non troppo distante da essa. Tra gli svantaggi, quelli di non costituire un modello sufficientemente formale da essere verificabile, e di non fornire astrazioni utili a rappresentare esplicitamente la comunicazione, la sincronizzazione e la concorrenza tra i processi.

Allo scopo di supplire alle debolezze menzionate, si è definito un procedimento di trasposizione di una specifica SDL in una rappresentazione in Reti di Petri posti/ transizioni, in quanto queste costituiscono un modello formale, che dispone di buoni risultati teorici e di algoritmi di analisi (cfr. [1], [2], [19]). La rappresentazione in Reti di Petri mantiene la struttura di controllo dei singoli processi, evidenziandone allo stesso tempo gli aspetti di comunicazione.

3. RAPPRESENTAZIONE IN RETI DI PETRI DI SPECIFICHE SDL

La trasposizione da SDL a Reti di Petri viene fatta, per ogni processo singolarmente, secondo le seguenti regole:

- ogni transizione SDL diventa una transizione della rete;
- lo stato SDL di partenza (di arrivo) di una transizione diventa un posto di ingresso (di uscita) della corrispondente transizione della rete;
- i segnali d'ingresso (d'uscita) di una transizione SDL diventano posti d'ingresso (d'uscita) della corrispondente transizione della rete. Segnali d'ingresso parametrici sono rappresentati con più posti, uno per ogni valore dei parametri;
- una decisione SDL con N rami d'uscita è rappresentata con N transizioni della rete, aventi in comune i posti d'ingresso corrispondenti a stato e input SDL, e ognuna avente in aggiunta un posto d'ingresso corrispondente a un valore della decisione;

- le operazioni e i segnali di temporizzazione (time-out) sono ignorati;
- i posti della rete sono etichettati con i corrispondenti nomi SDL;
- la marcatura iniziale della rete è assegnata attribuendo una marca al posto corrispondente allo stato SDL iniziale.

Nella rete ottenuta con queste regole, le operazioni sulle variabili interne e il contenuto delle decisioni non sono esplicitamente rappresentate. Ciò è congruente con il fatto che in SDL queste informazioni possono essere espresse in modo informale e non dettagliato, corrispondente a un livello di astrazione in cui i dati interni del processo non sono completamente specificati. Conseguentemente, nella rete i conflitti tra transizioni dovuti alle decisioni sono rappresentati e risolti non deterministicamente.
Similmente, la non rappresentazione esplicita dei segnali di temporizzazione comporta una risoluzione dei conflitti in modo non deterministico, anzicchè su base temporale.

Per quanto detto, sono le proprietà funzionali e di struttura di controllo dei processi SDL che si preservano nella rappresentazione in rete e sono sottoposti a verifica. Proprietà risultanti dalla durata relativa delle operazioni non sono rappresentate e non sono analizzate. Si ottiene cioè un verifica di tipo funzionale, che non include aspetti relativi alle prestazioni.

4. VERIFICA DI PROPRIETÀ

Le singole reti ottenute dai processi SDL possono essere composte insieme (come più in dettaglio si vedrà più avanti) ottenendo un'unica rete, rappresentante l'insieme dei processi con le loro comunicazioni, che può essere sottoposta a verifica.

Scopo della verifica è quello di individuare eventuali incompletezze, ambiguità o incongruenze nella specifica dei processi SDL, in particolare per quel che concerne la loro comunicazione. Ciò è utile in quanto la descrizione in processi, se da un canto favorisce la decomposizione del sistema e la specifica dettagliata di ogni suo componente, rende d'altro canto difficile confrontare i comportamenti di ciascun processo con quelli degli altri ai fini di garantire il corretto funzionamento dell'insieme.
Situazioni non desiderate possono presentarsi, ad esempio, quando i segnali previsti in ricezione da un processo in uno stato non sono previsti in spedizione da altri processi negli stati opportuni, e viceversa; oppure quando l'evoluzione dei processi porta a condizioni di blocco (deadlock) o di ciclo senza uscita (livelock).

Queste situazioni di interazione non corretta tra i processi possono in generale essere ricondotte alla verifica, nella rete che rappresenta la composizione dei processi, delle proprietà di reiniziabilità e ciclicità, limitatezza e vivezza della rete. Tali proprietà richiedono un'analisi di raggiungibilità, ovvero la costruzione del grafo delle marcature che rappresenta tutte le possibili sequenze di scatto della rete.

In aggiunta, la costruzione del grafo delle marcature consente di derivare in modo automatico insiemi sistematici di sequenze di messaggi che coprono le specifiche secondo un prestabilito criterio di copertura, e che possono quindi essere usate come piani di prova per il testing dei programmi implementati a partire da quelle specifiche.

Un lavoro precedentemente sviluppato presso la Direzione Centrale Ricerca e Sviluppo della Italtel Società Italiana Telecomunicazioni, nell'ambito del Progetto Finalizzato Informatica del CNR, ha portato a definire una metodologia, e a realizzare un insieme di strumenti, per la verifica di specifiche e la generazione automatica di casi di prova, sulla base delle linee sopra indicate [13], [14]. La verifica dell'utilità del metodo e la messa a punto delle funzionalità degli strumenti sono state effettuate su casi reali [6], [16].

Nel corso di tale sperimentazione, si è riscontrato un problema di prestazioni nell'uso degli strumenti, data la complessità dell'analisi di raggiungibilità per reti molto grandi, quali sono quelle che risultano da specifiche di sistemi reali. Ad esempio, in figura 1 è rappresentato lo schema dei processi che costituiscono la parte applicativa telefonica del livello 4 del Sistema di Segnalazione CCITT n. 7 [4]. La specifica SDL di questi processi consiste di 60 pagine, e la rappresentazione in reti di Petri risulta in reti aventi un totale di circa 300 posti e 1000 transizioni.

Ciò ha condotto a indagare ulteriori possibilità di affinare e incrementare l'applicabilità del metodo sviluppato. Queste indagini sono state rivolte da un lato verso l'uso di reti a più alto livello [8], e dall'altro verso l'identificazione di procedimenti di riduzione delle reti che mantenessero le proprietà da verificare. Nel seguito si tratteranno questi ultimi, e la relativa implementazione nel linguaggio Prolog.

5. METODO DI RIDUZIONE

La possibilità di ridurre le dimensioni di una rete corrispondente a un insieme di processi, allo scopo di facilitarne l'analisi automatica, si basa sul criterio di partizionare l'insieme dei processi di un sistema in sottoinsiemi, e di isolare ogni sottosistema così ottenuto dal resto del sistema, analizzandone le proprietà di congruenza interna indipendentemente dal suo contesto, cioè dalle interazioni con gli altri processi. Poichè i sottosistemi possono essere individuati a diversi livelli di aggregazione, ci si riconduce a una se-

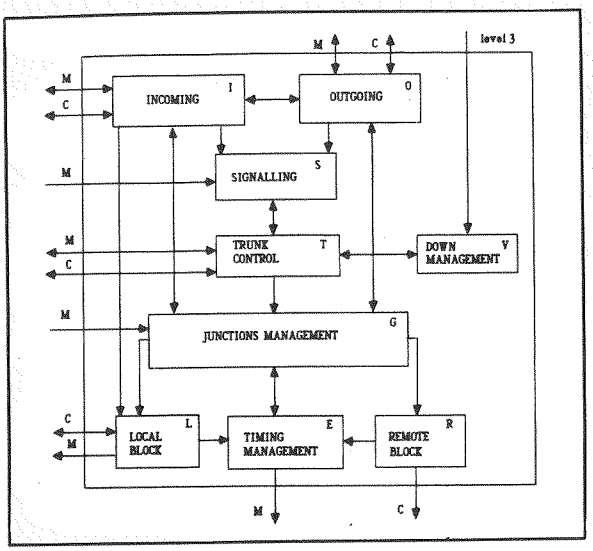


Fig. 1 Schema dei processi del sistema CCITT n. 7

rie di analisi parziali che danno risultati utili, pur senza inferirne un risultato globale che va visto come compito aggiuntivo. A scopo esemplificativo, in riferimento al sistema di figura 1, si considererà nel seguito il sottisistema costituito dai quattro processi G,L,E,R.

Isolando un sottosistema, nei processi componenti alcuni degli stati e delle transizioni, che tengono conto delle interazioni con i processi esterni al sottosistema considerato, diventano ridondanti. Poichè ogni processo è una macchina sequenziale a meno delle comunicazioni con gli altri, si possono applicare procedimenti simili a quelli di minimizzazione degli automi a stati finiti per eliminare tali ridonanze, mantenendo inalterato il comportamento di ciascuno di essi rispetto agli altri processi appartenenti al sottosistema considerato [7].

Una volta così ridotte le reti dei singoli processi, queste possono essere composte insieme facendo coincidere i posti che rappresentano segnali corrispondenti, cioè che appaiono in uscita da un processo e in ingresso a un altro. La rete composta rappresenta il sottosistema considerato; in essa compaiono solo comunicazioni interne, che si svolgeranno a partire dalla marcatura iniziale in cui ogni processo ha una marca nel posto corrispondente allo stato iniziale. La rete può quindi essere analizzata rispetto alle proprietà menzionate nel paragrafo precedente.

6. MOTIVAZIONI E CARATTERISTICHE DELL'IMPIEGO DI PROLOG PER L'IMPLEMENTAZIONE

Il ricorso alla programmazione logica [12], [11] e l'utilizzo di Prolog [5] come linguaggio di implementazione di uno strumento che realizza il metodo di riduzione sono stati indotti dalla volontà di verificare le seguenti ipotesi di lavoro:

- l'interpretazione di un programma logico come insieme di regole di produzione di una grammatica può facilitare la realizzazione di un traduttore, quale quello da SDL a Reti di Petri;
- la struttura a clausole di un programma Prolog è aderente al metodo di riduzione, che è organizzato a regole;
- la rappresentazione di una rete di Petri con un programma logico consente di vedere l'analisi delle proprietà della rete come derivazione di conseguenze logiche da premesse (fatti) concernenti la rete descritta;
- i tre aspetti precedenti possono essere ricondotti ad un unico formalismo logico usato in modo uniforme, malgrado l'apparente diversità dei sottoproblemi in esame;
- il Prolog si presta bene per una veloce implementazione di un prototipo dello strumento che, a seguito di una successiva sperimentazione, possa essere facilmente modificato ed esteso;
- oltre che per le sue caratteristiche ad alto livello, il Prolog può essere sfruttato per la prototipazione rapida mediante la costruzione di una libreria di moduli di base, da reimpiegare nei diversi programmi che contribuiscono alla funzionalità complessiva dello strumento.

Nel seguito si descrivono alcuni aspetti dell'implementazione che corrispondono a queste caratteristiche. Essa è stata fatta in C-Prolog, su workstation SUN-I con sistema operativo Unix.

6.1 Traduzione

L'interpretazione di un programma logico come sistema di regole di produzione della grammatica di un linguaggio, per il parsing e la traduzione di frasi di quel linguaggio, è stata evidenziata e sviluppata in [17], [21], a seguito di precedenti lavori di Colmerauer sull'elaborazione del linguaggio naturale, da cui, come è noto, è originato lo stesso Prolog.

Il linguaggio sorgente viene descritto mediante le Definite Clause Grammars (DCG), dove una regola di produzione contiene nella parte sinistra il simbolo non terminale da definire, e nella parte destra di definizione una successione di elementi separati da virgole. Questi elementi possono essere zero o più simboli terminali (termini Prolog inclusi in una lista, in parentesi quadre), simboli non terminali, e sottomete Prolog (queste ultime in parentesi graffe). Ai simboli non terminali possono essere forniti argomenti, che costituiscono strutture in cui costruire progressivamente l'albero sintattico durante il parsing.

Una produzione espressa in questa notazione può facilmente essere espansa in clausola Prolog: parte sinistra e parte destra della produzione diventano ri-

spettivamente testa e coda della clausola; un simbolo non terminale di molteplicità N diventa un predicato dello stesso nome a N+2 argomenti, i primi N uguali a quelli originari e i due aggiuntivi per rappresentare i punti di inizio e di fine della sottostringa corrispondente; le sottomete rimangono tali.

L'insieme delle clausole così ottenute opera, in base alle regole standard di ricerca e di computazione dell'interprete Prolog, come un parser ricorsivo discendente, con scansione della stringa d'ingresso da sinistra a destra e gestione delle alternative mediante il meccanismo di backtracking.

L'albero sintattico costruito con gli argomenti dei simboli non terminali può poi essere utilizzato per la generazione della stringa di uscita, mediante un altro insieme di regole, anch'esse espresse in procedure Prolog, che effettuano la trasposizione nelle strutture del linguaggio di destinazione.

Le DCG presentano quindi il vantaggio, rispetto alle usuali grammatiche BNF, di consentire la costruzione, durante il parsing, di strutture ad albero il più possibile corrispondenti a quelle d'uscita e non vincolate dalla struttura della grammatica, e di permettere l'inclusione nella grammatica di condizioni aggiuntive, che possono anche esprimere dipendenze dal contesto. Inoltre, la loro espansione in programma Prolog è molto semplice, ed è automatica in alcune implementazioni di Prolog, come il C-Prolog utilizzato nel lavoro qui descritto.

In fig. 2a è riportata la descrizione in BNF estesa dalla parte iniziale della grammatica di SDL, e in figura 2b la corrispondente descrizione in DCG, che costituisce parte dell'analizzatore sintattico. Come esempio di traduzione, sono riportati: in figura 3a il testo della specifica SDL del processo R; in figura 3b l'albero di parsing ottenuto dall'analisi sintattica; in figura 3c la rete risultante dalla traduzione, secondo le regole descritte nel paragrafo 3. La rete è espressa come insieme di fatti ognuno dei quali rappresenta una transizione, con lo stesso funtore principale e tre argomenti che danno rispettivamente la lista dei posti d'ingresso, il nome della transizione, e la lista dei posti d'uscita.

6.2 Riduzioni

Il procedimento di riduzione viene effettuato in due passi. Nel primo passo si cancellano i posti che rappresentano segnali di input e output esterni al sottosistema considerato. Questo comporta che alcune transizioni diventano "nulle", cioè ininfluenti rispetto alla comunicazione tra i processi in esame.
Esse possono venire eliminate mediante un insieme di regole che, tenendo conto dei cambiamenti di stato del processo per effetto di tali transizioni e della presenza delle altre transizioni, preservano l'insieme delle sequenze di segnali di comunicazione rappre-

```
<SDL block definition>   ::=  BLOCK <identifier> <end> <SDL process
                              definition> {<SDL process definition>} ENDBLOCK
                              <identifier>;
<SDL process definition> ::=  PROCESS <identifier> <end> <state body> {<state
                              body>} ENDPROCESS <identifier>;
<state body>             ::=  STATE <identifier> <end> <input> {<input>}
                              ENDSTATE <identifier>;
```

Fig. 2a BNF della parte iniziale della grammatica di SDL

```
/*  Predicato di piu' alto livello: definisce l'entita' 'blocco' dell'S.
    D. L.; un blocco consiste in una serie di uno o piu' processi.
*/
'SDL block definition'(Parse_Tree) -->
        { read('BLOCK') },              [ 'BLOCK' ],
        { read(Identifier) },           identifier(Identifier),
        { read(Token) },                end(Token),
        { read('PROCESS') },            [ 'PROCESS' ],
        'one or more SDL process definitions'(Processes_List),
        { Parse_Tree =.. ['BLOCK' | Processes_List] },
        [ 'ENDBLOCK' ],
        { read(Identifier) },           identifier(Identifier),
        { read(;) },                    [ ; ].

'one or more SDL process definitions'([Process | Tail]) -->
        'SDL process definition'(Process), { read(Identifier) },
        'one or more SDL process definitions_1'(Tail, Identifier).

'one or more SDL process definitions_1'(Tail, 'PROCESS') -->
        { ! }, 'one or more SDL process definitions'(Tail).

'one or more SDL process definitions_1'([], 'ENDBLOCK') --> [].

/*  Un processo S. D. L. consiste in uno o piu' stati.
*/
'SDL process definition'(Process) -->
        { read(Identifier) },           identifier(Identifier),
        { read(Token_1) },              end(Token_1),
        { read('STATE') },
        'one or more state bodies'(States_List),
        { Process =.. ['PROCESS' | States_List] },
        [ 'ENDPROCESS' ],
        { read(Identifier) },           identifier(Identifier),
        { read(Token_2) },              end(Token_2).

'one or more state bodies'([State | Tail]) -->
        'state body'(State), { read(Identifier) },
        'one or more state bodies_1'(Tail, Identifier).

'one or more state bodies_1'(Tail, 'STATE') -->
        { ! }, 'one or more state bodies'(Tail).

'one or more state bodies_1'([], 'ENDPROCESS') --> [].

/*  Uno stato S. D. L. consiste in uno o piu' inputs.
*/
'state body'('STATE'(State_Name, Inputs_List)) -->
        [ 'STATE' ],
        { read(State_Name) },           identifier(State_Name),
        { read(Token) },                end(Token),
        { read('INPUT') },              [ 'INPUT' ],
        { read(Input_Name) },           identifier(Input_Name),
        'one or more inputs'(State_Name,Input_Name,Inputs_List).
```

Fig. 2b Corrispondente definizione DCG che è parte del parser

```
BLOCK B;
    PROCESS R 'blocco remoto';
        STATE R0 'riposo';
            INPUT GR_UBR;
            OUTPUT RC_UBA;
            NEXTSTATE R0 'riposo';
            INPUT GR_BLR;
            OUTPUT RC_BLA;
            OUTPUT RE_BLR;
            NEXTSTATE R1 'blocco remoto';
        ENDSTATE R0;
        STATE R1 'blocco remoto';
            INPUT GR_UBR;
            OUTPUT RC_UBA;
            OUTPUT RE_UBR;
            NEXTSTATE R0 'riposo';
            INPUT GR_BLR;
            OUTPUT RC_BLA;
            NEXTSTATE R1 'blocco remoto';
        ENDSTATE R1;
    ENDPROCESS R;
ENDBLOCK B;
```

Fig. 3a Specifica SDL del processo R

```
'STATE'(
'R0',
[
'INPUT'('GR_UBR', 'OUTPUT'(['RC_UBA']), 'NEXTSTATE'('R0')),
'INPUT'('GR_BLR', 'OUTPUT'(['RC_BLA', 'RE_BLR']), 'NEXTSTATE'('R1'))
]
).
'STATE'(
'R1',
[
'INPUT'('GR_UBR', 'OUTPUT'(['RC_UBA', 'RE_UBR']), 'NEXTSTATE'('R0')),
'INPUT'('GR_BLR', 'OUTPUT'(['RC_BLA']), 'NEXTSTATE'('R1'))
]
).
```

Fig. 3b Albero sintattico

```
net_transition(['R0', 'GR_UBR'], t01, ['R0', 'RC_UBA']).
net_transition(['R0', 'GR_BLR'], t02, ['R1', 'RC_BLA', 'RE_BLR']).
net_transition(['R1', 'GR_UBR'], t03, ['R0', 'RC_UBA', 'RE_UBR']).
net_transition(['R1', 'GR_BLR'], t04, ['R1', 'RC_BLA']).
```

Fig. 3c Rete R risultante dalla traduzione

sentate nella rete ridotta rispetto a quelle della rete di partenza.

Sono state definite 15 regole che consentono queste trasformazioni. Una di esse, che ne esemplifica la struttura, è rappresentata dallo schema della figura 4 (la transizione non etichettata è quella nulla da eliminare, applicabile quando sono verificate le seguenti condizioni:

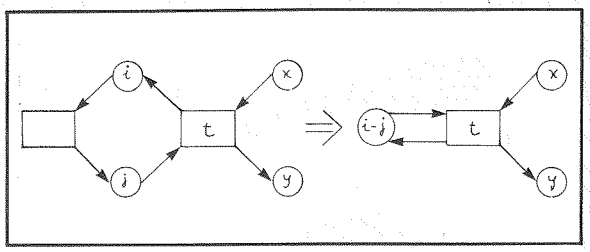


Fig. 4 Un esempio di regola di eliminazione di transizioni nulle

- i non ha transizioni in ingresso da altri posti nè in uscita verso altri posti; non sono ammesse transizioni in loop su questo posto;
- j ha transizioni in ingresso da altri posti e in uscita verso altri posti; sono ammesse transizioni il loop su questo posto;
- possono esserci altre transizioni, nulle o non nulle, da j a i, ma non da i a j.

In figura 5 è riportata la parte di procedura Prolog che realizza questa regola. Vi si può notare come la modularità delle clausole favorisce la trasparenza del programma rispetto alla descrizione informale delle regole stesse.

Dopo questo primo passo di eliminazione delle transizioni nulle, un secondo passo effettua una ulteriore minimizzazione, che lascia sempre invariate le sequenze di segnali di comunicazione, con procedimenti che non verranno qui ulteriormente discussi (sono descritti in [7] e [9]). Come esempio di risultato, la rete di figura 6a, ottenuta dalla traduzione della specifica SDL del processo E, viene ridotta a quella di figura 6b; è possibile vedere che essa presenta, rispetto a quella di partenza, le stesse sequenze di comunicazione rispetto ai tre altri processi R, G, L considerati (i nomi dei segnali di comunicazione sono quelli che hanno un prefisso di due lettere, che denotano rispettivamente i processi mittente e destinatario).

Il procedimento di riduzione può essere iterato. Ad esempio, si vogliano analizzare, all'interno del sottosistema dei quattro processi, i comportamenti dei processi considerati a coppie. Per ogni coppia, la riduzione si effettua a partire dalla cancellazione dei posti di comunicazione esterna rispetto alla coppia. In figura 7 sono riportate le reti ottenute con alcune di tali riduzioni "di secondo livello".

6.3 Analisi

La composizione delle reti ridotte si effettua semplicemente giustapponendo (mettendo insieme nel data base Prolog) i relativi fatti. Di nuovo, la modularità della rappresentazione Prolog rende immediata l'operazione. La rete composta può presentare:

```
/*   Ottava regola di eliminazione delle azioni nulle.
     Regola R4-5.
*/

     delete_all_null_actions_2([fs3_transition(I,T,O)|L2],List) :-
          length(I,I), length(O,1), find_SDL_state(X,I), find_SDL_state(Y,O),
          member(fs3_transition(M,T1,N),List),
          T1\== T,
          find_SDL_state(Y,M), find_SDL_state(X,N),
          length(M,Length_M), length(N,Length_N),
          (Length_M > 1 ; Length_N > 1),
/*        i non ha transizioni entranti da altri nodi
*/        find_all(fs3_transition(A1,T2,B1),
                        (member(fs3_transition(A1,T2,B1),List),
                         find_SDL_state(X,B1), find_SDL_state(Y1,A1),
                         Y1\== X, Y1\== Y, T2\== T1),
                                        List_1),
          List_1 == [],
/*        i non ha transizioni uscenti verso altri nodi
*/        find_all(fs3_transition(A2,T3,B2),
                        (member(fs3_transition(A2,T3,B2),List),
                         find_SDL_state(X,A2), find_SDL_state(Y2,B2),
                         Y2\== X, Y2\== Y),
                                        List_2),
          List_2 == [],
/*        i non deve presentare transizioni in loop
*/        find_all(fs3_transition(A5,T6,T5),
                        (member(fs3_transition(A5,T6,B5),List),
                         find_SDL_state(X,A5), find_SDL_state(X,B5),
                         T6\== T),
                                        List_5),
          List_5 == [],
/*        j ha transizioni entranti da altri nodi
*/        find_all(fs3_transition(A3,T4,B3),
                        (member(fs3_transition(A3,T4,B3),List),
                         find_SDL_state(Y,B3), find_SDL_state(X3,A3),
                         X3\== X, X3\== Y),
                                        List_3),
          List_3\== [],
/*        j ha transizioni uscenti
*/        find_all(fs3_transition(A4,T5,B4),
                        (member(fs3_transition(A4,T5,B4),List),
                         find_SDL_state(Y,A4), find_SDL_state(Y4,B4),
                         Y4\== X, Y4\== Y, T5\== T1),
                                        List_4),
          List_4\== [],
          !,
          nl,nl,tab(15),write('Regola R4-5.'),
          nl,nl,tab(15),write('Transizione interessata:'),
          tab(5),writeq(fs3_transition(I,T,O)),write('.),
/*        aggiornamento della lista di servizio e della lista contenente
          le transizioni della rete

*/        efface(fs3_transition(I,T,O),List,List1),
          nodes_concatenation([X,Y],Conc),
          updating(L2,X,Y,Conc,New), updating(List1,X,Y,Conc,New_List),
          nl, writing(New), nl, writing(New_List),
          delete_all_null_actions_2(New,New_List).
```

Fig. 5 Clausola Prolog per una regola di eliminazione di transizioni nulle

```
net_transition(['E0', 'LE_BLO/RE_BLR'], t01, ['E4', 'EG_TEST']).
net_transition(['E0', 'GE_LIB'], t02, ['E0']).
net_transition(['E1', 'LE_BLO/RE_BLR'], t03, ['E1']).
net_transition(['E1', 'GE_LIB'], t04, ['E2']).
net_transition(['E1', 'LE_UBA/RE_UBR'], t05, ['E6', 'EG_TEST']).
net_transition(['E2', 'LE_BLO/RE_BLR'], t06, ['E2']).
net_transition(['E2', 'GE_LIB'], t07, ['E2']).
net_transition(['E2'], t08, ['E3', 'EM_ALR4']).
net_transition(['E2', 'LE_UBA/RE_UBR'], t09, ['E5', 'EG_TEST']).
net_transition(['E3', 'LE_BLO/RE_BLR'], t10, ['E3']).
net_transition(['E3', 'GE_LIB'], t11, ['E3']).
net_transition(['E3', 'LE_UBA/RE_UBR'], t12, ['E7', 'EG_TEST']).
net_transition(['E4', 'GE_RIS:I=yes'], t13, ['E1']).
net_transition(['E4', 'GE_RIS:I=no'], t14, ['E2']).
net_transition(['E5', 'GE_RIS:B=yes'], t15, ['E2']).
net_transition(['E5', 'GE_RIS:B=no'], t16, ['E0']).
net_transition(['E6', 'GE_RIS:B=yes'], t17, ['E1']).
net_transition(['E6', 'GE_RIS:B=no'], t18, ['E0']).
net_transition(['E7', 'GE_RIS:B=yes'], t19, ['E3']).
net_transition(['E7', 'GE_RIS:B=no'], t20, ['E0', 'EM_FAL4']).
```

Fig. 6a Rete E risultante dalla traduzione

```
reduced_net_transition(['E0', 'LE_BLO/RE_BLR'], t01, ['E4', 'EG_TEST']).
reduced_net_transition(['E0', 'GE_LIB'], t02, ['E0']).
reduced_net_transition(['E4', 'GE_RIS:I=yes'], t13, ['E1-E2-E3']).
reduced_net_transition(['E4', 'GE_RIS:I=no'], t14, ['E1-E2-E3']).
reduced_net_transition(['E1-E2-E3', 'GE_LIB'], 't04-t07-t11', ['E1-E2-E3']).
reduced_net_transition(['E1-E2-E3', 'LE_BLO/RE_BLR'], 't03-t06-t10', ['E1-E2-E3']).
reduced_net_transition(['E1-E2-E3', 'LE_UBA/RE_UBR'], 't05-t09-t12', ['E5-E6-E7','EG_TEST']).
reduced_net_transition(['E5-E6-E7', 'GE_RIS:B=yes'], 't15-t17-t19', ['E1-E2-E3']).
reduced_net_transition(['E5-E6-E7', 'GE_RIS:B=no'], 't16-t18-t20', ['E0']).
```

Fig. 6b Rete E ridotta

```
Rete R ridotta rispetto ad E.

reduced_net_transition(['R0'], 1, ['R1', 'RE_BLR']).
reduced_net_transition(['R1'], 2, ['R0', 'RE_UBR']).

Rete E ridotta rispetto a R.

reduced_net_transition(['E0', 'RE_BLR'], 3, ['E1']).
reduced_net_transition(['E1', 'RE_BLR'], 4, ['E1']).
reduced_net_transition(['E1', 'RE_UBR'], 5, ['E0']).
reduced_net_transition(['E1', 'RE_UBR'], 6, ['E1']).

Rete L ridotta rispetto a G.

reduced_net_transition(['L0', 'GL_BLO'], 1, ['L1']).
reduced_net_transition(['L1', 'GL_BLO'], 2, ['L1']).
reduced_net_transition(['L1', 'GL_UBL'], 3, ['L0']).

Rete G ridotta rispetto a L.

reduced_net_transition(['G0'], 4, ['G1', 'GL_BLO']).
reduced_net_transition(['G1'], 5, ['G0', 'GL_UBL']).
```

Fig. 7 Reti ridotte al secondo livello

- parallelismo su sottoinsiemi di transizioni;
- conflitti tra transizioni che hanno posti d'ingresso corrispondenti a diverso valore delle decisioni: vengono risolti, come si è detto prima, non deterministicamente;
- conflitti tra transizioni che hanno posti d'ingresso corrispondenti ai segnali interni e transizioni che non li hanno (perchè i posti corrispondenti a segnali esterni o di time-out sono stati rimossi); corrispondentemente al meccanismo della coda SDL, vengono risolti dando priorità alle prime.
  Questo fa sì che non si generano contatti sui posti della rete, ovvero ogni posto riceverà al più una marca.

Il programma di analisi di raggiungibilità della rete realizza una regola di scatto delle transizioni basata sulla suddetta priorità; esso inoltre riconosce le transizioni concorrenti. Il programma costruisce il grafo delle marcature della rete e lo rappresenta mediante una serie di fatti, ognuno relativo allo scatto di una transizione (o di un insieme di transizioni concorrenti), con tre argomenti che danno la marcatura di par-

tenza come lista dei posti marcati prima dello scatto, la transizione scattata, e la marcatura raggiunta come lista dei posti marcati dopo lo scatto. Ad esempio, per le composizioni delle due coppie di reti di figura 7, i grafi delle marcature sono riportati in figure 8a e 8b.

Il programma di analisi, sulla base di tale rappresentazione, verifica quindi le proprietà di reinizziabilità, ciclicità e vivezza della rete, fornendo i relativi messaggi. Nell'esempio, il risultato riportato in figura 8a è positivo mentre quello di figura 8b mostra che la rete non è viva. Questo è dovuto ad un errore nella specifica del processo L, che ammette di ricevere un numero indefinito di volte consecutive il segnale GL-BLO, mentre il processo G manda alternativamente i segnali GL-BLO e GL-UBL.
L'incoerenza si risolve eliminando la transizione non viva della rete, e le corrispondenti transizioni nella specifica SDL.

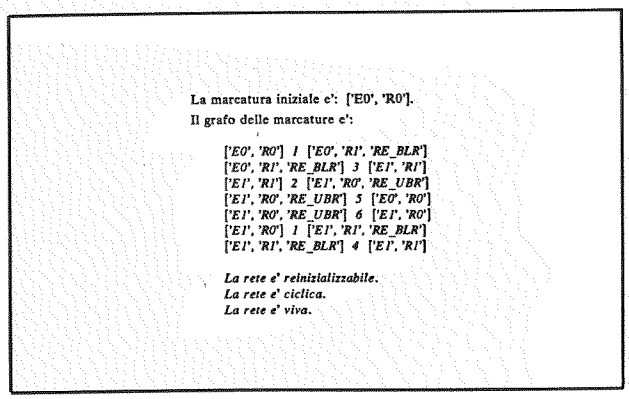


Fig. 8a Risultati dell'analisi per la rete composta da E ridotta rispetto ad R ed R ridotta rispetto ad E

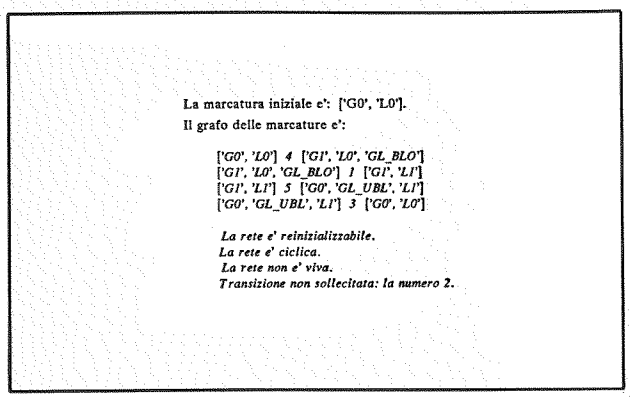


Fig. 8b Risultati dell'analisi per la rete composta da G ridotta rispetto ad L ed L ridotta rispetto a G

7. VALUTAZIONI E CONCLUSIONI

La realizzazione dello strumento descritto ha consentito di confermare le ipotesi di lavoro menzionate nel paragrafo 6. Si riassumono qui i risultati ottenuti con l'utilizzo di Prolog, anche alla luce di una precedente implementazione (per la traduzione e l'analisi, non per la riduzione) in linguaggi procedurali (Pascal, C e shell di Unix):

- i programmi Prolog risultano molto compatti come dimensioni, e richiedono un minore tempo di sviluppo; quelli sopra descritti consistono di 350 procedure, per un totale di circa 900 clausole;
- la modularità delle clausole nei programmi Prolog agevola le modifiche; questa caratteristica è stata riscontrata in diverse modifiche fatte sulle regole di riduzione e sulle regole di scatto;
- la riusabilità di procedure Prolog in più programmi è possibile e riduce ulteriormente il lavoro; si è sviluppata una libreria con numerose procedure di base, alcune delle quali utilizzate varie volte in diversi programmi;
- si è ottenuta uniformità di trattamento delle tre componenti dello strumento, che sono di natura diversa come funzionalità ma hanno in comune la strutturazione a regole del problema; tale uniformità risulta in facile e trasparente documentazione dei programmi;
- il Prolog interpretato presenta dei limiti di prestazioni. Si sono tuttavia individuati, e impiegati con molto profitto, diversi possibili modi per far fronte ad essi, in particolare per ovviare alle criticità delle computazioni ricorsive troppo ampie.

Complessivamente, le caratteristiche suddette confermano il Prolog come un ottimo strumento di prototipazione di software.

In conclusione, si ritiene che il metodo di riduzione implementato possa essere utilizzato non solo per sistemi specificati in SDL, ma per tutti i sistemi rappresentabili in reti di Petri strutturate a processi sequenziali comunicanti, problematica che si ha ad esempio anche nel campo dei protocolli di comunicazione (cfr. [22]). La potenza e semplicità delle Definite Clause Grammars possono tornare utili nell'eventuale costruzione di traduttori da altri particolari linguaggi di specifica a reti di Petri, su cui applicare i procedimenti di riduzione e di analisi.

8. RINGRAZIAMENTI

I programmi Prolog sono stati sviluppati da F. Furlan come lavoro di tesi di laurea svolto in stage presso la Italtel Società Italiana Telecomunicazioni, che si ringrazia.

9. BIBLIOGRAFIA

[1] Brams G.W., RESEAUX DE PETRI: THEORIE ET PRATIQUE, Masson, 1983, Trad. it. Masson Italia, 1985

[2] Brauer W. (ed.), NET THEORY AND APPLICATIONS, Lecture Notes in Computer Science n. 84, Springer Verlag, 1980

[3] CCITT, REPORT TO THE PLENARY ASSEMBLY - PART III.11: RECOMMENDATIONS ON THE FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL) (Recommendations Z.100 to Z.104). VIII Plenary Assembly, Malaga-Torremolinos, (Document AP VIII-85-E), 1984

[4] CCITT, SPECIFICATION OF CCITT COMMON CHANNEL SIGNALLING SYSTEM n. 7 (part III B of the Report of Study Group XI to the Plenary Assembly), VII Plenary Assembly (document AP VII-18-E), Geneva 1980

[5] Clocksin W.F., Mellish C.S., PROGRAMMING IN PROLOG, Springer Verlag, 1981

[6] Comparin G., Lanzarone G.A., Panzeri W., Torgano A., ANALYSIS OF SDL SPECIFICATIONS USING PETRI NETS TECHNIQUES, Procs. 2nd SDL Users and Implementors Forum, Helsinki, march 11-15, 1985

[7] Comparin G., Lanzarone G.A., Lautenbach K., Pagnoni A., Panzeri W., Torgano A., GUIDELINES ON USING NET ANALYSIS TECHNIQUES WITH LARGE SPECIFICATIONS, Procs 6th European Workshop on Applications and Theory of Petri Nets, Espoo (Finland), june 26-28, 1985. In corso di pubblicazione in: "Advances in Petri Nets 85", Springer Verlag

[8] De Cindio F., Lanzarone G.A., Torgano A., A PETRI NET MODEL OF SDL, Procs 5th European Workshop on Applications and Theory of Petri Nets, Aarhus (Denmark), june 27-29, 1984. Altre versioni in: Atti del Congresso Annuale AICA 84, Roma 24-26 ott. 1984, e in: Management e Informatica, maggio 1985

[9] Furlan F., SVILUPPO DI METODI PER L'ANALISI DI SISTEMI DI TELECOMUNICAZIONI E LORO REALIZZAZIONE NEL LINGUAGGIO PROLOG, Tesi di laurea in Fisica, Università degli Studi di Milano, a.a. 1984-85, relatori G.A. Lanzarone, G. Degli Antoni, F. De Cindio

[10] Hoare C.A.R., COMMUNICATING SEQUENTIAL PROCESSES, Prentice Hall, 1985

[11] Hogger C.J., INTRODUCTION TO LOGIC PROGRAMMING, Academic Press, 1984

[12] Kowalski R., LOGIC FOR PROBLEM SOLVING, North-Holland, 1979

[13] Lanzarone G.A., AUTOMATIC SPECIFICATION VERIFICATION AND TEST-CASE GENERATION FOR REAL-TIME SYSTEMS, Procs. 3rd European Workshop on Applications and Theory of Petri Nets, Varenna, sept. 27-30, 1982

[14] Lanzarone G.A., AUTOMATIC FUNCTIONAL TEST-CASE GENERATION FOR REAL-TIME CONTROL SYSTEMS, 6th Conf. on Software Engineering (poster session Procs.), Tokyo, sept. 13-16, 1982

[15] Martin R., Memmi G., SPECIFICATION AND VALIDATION OF SEQUENTIAL PROCESSES COMMUNICATING BY FIFO CHANNELS, Procs. Int. Conf. on Software Engineering for Telecommunication Switching Systems, Warwick, 1981

[16] Mattavelli R., IMPOSTAZIONE DI SPECIFICHE, IMPLEMENTAZIONE E PROVA DEL SISTEMA CCITT n. 7 (SEGNALAZIONE TELEFONICA SU CANALE COMUNE) SECONDO UNA METODOLOGIA BASATA SU SDL, Tesi di laurea in Fisica, Università degli Studi di Milano, a.a. 1983-84, relatori G. Marietti, G.A. Lanzarone

[17] Pereira F.C.N., Warren D.H.D., DEFINITE CLAUSE GRAMMARS FOR LANGUAGE ANALYSIS - A SURVEY OF THE FORMALISM AND A COMPARISON WITH AUGMENTED TRANSITION NETWORK, Artificial Intelligence, vol. 13, n. 3, may 1980, pp. 231-278

[18] Reisig W., DETERMINISTIC BUFFER SYNCHRONIZATION OF SEQUENTIAL PROCESSES, Acta Informatica, vol. 18, n. 2, 1982, pp. 117-134

[19] Reisig W., PETRI-NETZE, EINE EINFUHRUNG, Springer Verlag, 1982. Trad. it. "Le Reti di Petri", Mondadori (edizioni elettroniche), 1984

[20] Rockstrom A., Saracco R., SDL-CCITT SPECIFICATION AND DESCRIPTION LANGUAGE, IEEE Trans. on Comm., vol. COM-30, n. 6, june 1982

[21] Warren D.H.D., LOGIC PROGRAMMING AND COMPILER WRITING, Software Practice and Experience, vol. 10, 1980, pp. 97-125

[22] Yemini Y., Strom R., Yemini S. (eds), PROTOCOL SPECIFICATION, TESTING, AND VERIFICATION IV, North-Holland, 1985

RAPPRESENTAZIONE LOGICA DI TESTI DI LEGGE

Roberto Bertocchi, Monica Stagnoli

Dipartimento di Scienze dell'Informazione - Università di Milano

RIASSUNTO

Questo lavoro descrive un progetto di ricerca in corso dedicato alla formalizzazione dei testi di legge come programmi logici.
La rappresentazione di norme legali in termini logici chiarifica il contenuto delle leggi e permette la costruzione di sistemi software di vario genere.
Sistemi esperti di tipo legale possono essere utilizzati per applicare la legge, per verificare la coerenza e la completezza di un frammento di legislazione e per identificare implicazioni non volute prima che la legge sia emessa.
Il dominio legale è un campo di indagine interessante per l'Intelligenza Artificiale, poichè coinvolge un vasto insieme di tematiche come la rappresentazione della conoscenza, la formalizzazione di testi in linguaggio naturale e il ragionamento di senso comune.

ABSTRACT

This paper describes an ongoing research project devoted to the formalization of law texts as logic programs.
The representation of legal rules in logical terms makes clear the contents of law and allows to build different kinds of software systems.
Legal expert systems may be used to apply the law by providing advice or may help the law makers to check the soundness and the completeness of a piece of legislation and to identify unwanted implications before the law will be issued.
The legal domain is also and interesting field for AI research since it involves a wide set of challenging issues as knowledge representation, the formalization of natural language texts and commonsense reasoning.

1. INTRODUZIONE

Gran parte dell'attività delle organizzazioni è regolamentata da norme legali, la rappresentazione di tali regole in una forma comprensibile da un elaboratore permette la realizzazione di sistemi che utilizzano praticamente la conoscenza delle leggi limitando il ricorso ad esperti umani.

Sistemi software che incorporano la conoscenza di un insieme di norme giuridiche possono essere utilizzati sia nella fase di applicazione delle leggi che nella fase della loro stesura.
L'esempio più semplice di un sistema del primo tipo è costituito da un sistema che calcola automaticamente la paga di un impiegato tenendo conto delle detrazioni fiscali, previdenziali ed assistenziali previste dalla legge. Un esempio più complesso è costituito da un sistema di consultazione che, in presenza di determinate circostanze, suggerisce quali azioni sono legalmente permesse, vietate od obbligatorie. Un tale sistema può essere utilizzato da solo o far parte di un sistema più ampio come un sistema d'ufficio o un sistema di supporto alle decisioni.
Sistemi software possono altresì supportare l'attività del legislatore permettendo di identificare punti controversi e implicazioni non volute prima che la legge venga emessa, e di valutare le conseguenze sociali ed economiche provocate da una modifica dell'ordinamento legislativo vigente.

La progettazione di tali sistemi richiede la formalizzazione delle norme giuridiche contenute nei testi legi-

slativi e la loro rappresentazione mediante un modello rigoroso trattabile da un elaboratore.

Allo stato attuale non esiste una teoria consolidata ma sono state avanzate numerose proposte [4], [14]; in particolare diversi autori hanno suggerito di adottare la logica simbolica come formalismo per la rappresentazione e l'analisi della legge.
Questo approccio appare promettente per i numerosi vantaggi offerti da un formalismo logico. Per esplorare il potenziale della logica in ambito legale abbiamo formalizzato in Prolog un frammento della legge che regolamenta il funzionamento dell'Università. La nostra ricerca è rivolta ai seguenti obiettivi:

- verificare in che misura le clausole di Horn sono un formalismo adeguato per la rappresentazione di norme legislative e per la costruzione di sistemi esperti di tipo legale;
- identificare meccanismi per rendere agevole la traduzione dei testi legislativi in programmi logici;
- costruire un setting sperimentale nel quale possano essere analizzati problemi di rappresentazione di conoscenza e tentativi di soluzione possono essere sperimentati.

Nel seguito è descritto come un insieme di norme possono essere formalizzate in Prolog, sono indicate le linee metodologiche che sono state seguite e sono illustrati i diversi modi nei quali la formalizzazione può essere utilizzata.

2. RAPPRESENTAZIONE DI NORME GIURIDICHE

Una norma giuridica è una regola a cui devono attenersi i membri di una collettività. Una norma giuridica è espressa da un testo in linguaggio naturale.

L'utilizzo del linguaggio naturale genera sia un'ambiguità semantica, determinata dal significato delle parole, che un ambiguità sintattica, generata dalle relazioni tra le parole con le quali la norma è espressa.
Questo implica da un lato che una norma può essere interpretata in modi diversi, e, dall'altro che una stessa norma può essere espressa in forme diverse che possono differire per i termini utilizzati, le costruzioni sintattiche utilizzate per collegarli (attivo, passivo,...) e per i modi verbali impiegati (indicativo, imperativo,...).

È opportuno quindi distinguere:

- la struttura superficiale di una norma costituita dalla sua espressione linguistica;
- la struttura profonda di una norma cioè cosa la norma afferma.

La formalizzazione di una norma, data la molteplicità delle sue possibili formulazioni linguistiche, è quindi rivolta alla definizione della sua struttura profonda.
Una norma è usualmente descritta nei testi legislativi mediante la struttura

se S allora C

dove S denota una determinata situazione/circostanza/fatto e C denota la conseguenza causata dal verificarsi di S.

La connessione fra S e C è generalmente descritta per mezzo di verbi come potere, dovere, essere obbligato.

Numerose proposte per adottare la logica simbolica come strumento per l'analisi e la formulazione stessa delle leggi sono state avanzate da diversi studiosi del diritto [1], [16], [19].

Questo approccio appare promettente in quanto la logica fornisce un linguaggio formale che è particolarmente adatto per esprimere concetti di tipo definizionale che specificano relazioni e proprietà legali.
Una formulazione logica di un insieme di norme fornisce una rappresentazione di tipo dichiarativo con una semantica rigorosa ed, inoltre, la legislazione è spesso già espressa in una forma simile a quella logica e quindi una formalizzazione che mantenga la struttura del testo originale sarà più facile da verificare e da aggiornare in futuro al variare della legislazione.

Adottando un approccio di tipo logico l'espressione di una norma viene scomposta in proposizioni elementari e ricondotta ad una struttura nella quale uno o più antecedenti sono uniti ad un conseguente per mezzo di un'implicazione logica. In questo modo si ottiene una rappresentazione non ambigua delle relazioni logiche tra le proposizioni che costituiscono la norma. Il processo di rappresentazione non elimina però l'ambiguità semantica legata al significato della parole utilizzate e quindi una formalizzazione costituirà sempre una particolare interpretazione di una norma.

La disponibilità di interpreti efficienti per linguaggi basati su un sottoinsieme della logica del primo ordine (clausole di Horn) permette di utilizzare praticamente la rappresentazione della legge per costruire sistemi esperti di tipo legale. Infatti le clausole di Horn costituiscono la base del linguaggio di programmazione Prolog che permette di derivare le conseguenze logiche implicate da un insieme di assiomi.

Diverse esperienze di rappresentazione di leggi come programmi logici sono state effettuate all'Imperial College da [21], [22] e all'Università di Waterloo da [12]. Altre ricerche, basate su formalismi non logici, sono state condotte da McCarthy [15], Waterman e

Peterson [24] e Gardner [8].
La nostra ricerca è più vicina al primo approccio, il nostro obiettivo è individuare tecniche e metodologie per tradurre il contenuto dei testi legislativi in programmi logici.

3. LA LEGGE ANALIZZATA

La situazione che è stata presa in considerazione riguarda l'attribuzione di un insegnamento universitario vacante ad un docente.

I testi che contengono la normativa sono-
- la legge n. 382 dell'11/7/1980 (art. 9, 25, 29, 100, 114, 116)
- la legge n. 477 del 13/8/1984, (art. 1,3) che modificano, rispettivamente, gli art. 9 e 114 della legge 382
- la circolare interpretativa della legge n. 477 del 16/10/1984 riguardante il conferimento di supplenze.

L'attribuzione può essere effettuata secondo diverse modalità:
a) *per "sostituzione"*
   i.e. il docente, al posto del corso di cui è titolare, tiene il corso vacante
b) *per "secondo insegnamento"*
   i.e. il docente, oltre al proprio corso, è incaricato di tenere anche il corso vacante
c) *per "supplenza"*
   i.e. il docente sostituisce il titolare del corso

L'utilizzo di una modalità è in relazione alle caratteristiche del corso, dell'Università di appartenenza e del docente.

Si è scelto di rappresentare questo frammento di legge perchè esso presenta molte caratteristiche tipiche dei testi legislativi. La conoscenza è distribuita in diversi testi, contiene riferimenti a leggi precedenti ed è stata modificata più volte. Alcune norme sono definite esplicitamente mentre altre devono essere estratte da norme che regolano altre situazioni. La legge, oltre a contenere precise definizioni, esprime anche principi generali ed affermazioni di meta livello, e fa affidamento, in numerosi casi, a conoscenze implicite.

4. LA FORMALIZZAZIONE DELLA LEGGE

Poichè le norme che disciplinano l'attribuzione dell'insegnamento vacante sono distribuite in più leggi e, all'interno di ciascuna legge, in diversi articoli, la rappresentazione è stata effettuata secondo questi criteri metodologici:

a) dapprima è stato formalizzato separatamente ciascun articolo della legge 382,
b) le regole ottenute sono state coordinate, ed infine
c) la formalizzazione è stata modificata per tener conto della legge 477 e della circolare interpretativa.

La rappresentazione dei singoli articoli di legge è stata realizzata mediante l'analisi dei periodi che compongono il testo, al fine di determinare la struttura logica della/e norma/e espressa/e nel periodo.
Per individuare tale struttura si è scomposto il periodo nelle sue proposizioni costituenti e queste ultime sono state classificate come:

- azioni rilevanti;
- condizioni che le determinano;
- relazioni che sussistono tra le condizioni e le azioni.

La struttura logica della norma è stata quindi rappresentata con una regola Prolog. Ad esempio:

Art. 114, comma 1, 1 parte

> Fino all'espletamento della prima tornata dei giudizi di idoneità per professore associato i posti di insegnamento rimasti vacanti per qualsiasi ragione, semprechè per l'insegnamento che si intende ricoprire per supplenza sia stato richiesto il posto di ruolo, e per i quali sia comprovata l'impossibilità di chiamata di professori di ruolo, possono essere conferiti per supplenza esclusivamente a professori ordinari e straordinari, a professori associati ovvero a professori incaricati stabilizzati; della stessa materia o di materia affine, appartenenti alla stessa facoltà;...

è rappresentato dalla clausola Prolog:

```
Insegnamento puo_essere_attribuito_in_supplenza_a Docente :-  (1)
        non_espletate_tornate_giudizi_di_idoneita,            (2)
        Insegnamento e_vacante,                               (3)
        è_stato_righiesto_posto_di_ruolo_per Insegnamento,    (4)
        impossibilità_di_chiamata_per Insegnamento,           (5)
        Docente e_un_ord_o_straord_o_assoc_o_inc-stab,        (6)
        Insegnamento e_stessa_materia_o_affine_a_Docente.     (8)
```

Nella regola precedente il predicato (1) rappresenta l'azione regolamentata dalla legge, mentre i predicati (2)-(8) descrivono le condizioni che devono essere verificate affinchè tale azione possa essere eseguita.

Si noti che ogni predicato corrisponde ad un frammento del testo analizzato, ad esempio: "e_stato_richiesto_posto_di_ruolo_per Insegnamento" rappresenta la parte di testo:
"... semprechè per l'insegnamento che si intende ricoprire per supplenza sia stato richiesto il posto di ruolo..."

Ciascun predicato descrive una condizione elementare o complessa.

Condizioni complesse sono costituite dalla congiunzione/disgiunzione di condizioni e sono specificate per mezzo di altre regole.
Ad esempio:
la condizione "Insegnamento e_stessa_materia_o_affine_a Docente" è specificata dalla regola:

```
Insegnamento e_stessa_materia_o_affine_a Docente :-
        Insegnamento e_stessa_materia Docente;
        Insegnamento affine_a Docente.
```

La descrizione di una condizione in termini di altre condizioni è frequentemente determinata dalla presenza di definizioni contenute in altri contesti normativi. Ad esempio, le affinità tra gli insegnamenti sono definite in un altra parte della legge in termini di appartenenza allo stesso gruppo concorsuale e, inserite nella rappresentazione come fatti, permettono di specificae ulteriormente la condizione "Insegnamento affine_a Docente"

```
Insegnamento affine_a Docente :-
        Insegnamento appartiene_raggruppamento_concorsuale X,
        Docente insegna Corso,
        Corso appartiene_raggruppamento_concorsuale X.
```

La traduzione della seconda parte dell'art. 114 è stata più problematica:

Art. 114, comma 1, 2 parte

> ... Non possono comunque essere coperti per supplenza gli insegnamenti sdoppiati salvo che il numero degli esami sostenuti negli insegnamenti stessi nell'ultimo anno accademico sia superiore a 250 per ciascun corso attivato.

Questa norma descrive quando l'azione "conferimentodi supplenza" è proibita e dovrebbe essere rappresentata come:

```
Insegnamento e_uno_sdoppiamento AND esami_insuff_in Insegnamento ⊃ NOT (Insegnamento puo_essere_attribuito_in_supplenza_a Docente)
```

Ma poichè in Prolog non sono ammesse regole con conclusioni negative, per rappresentare la norma esistono due possibilità:

1) trasformare la formulazione della norma
2) esprimere la proibizione in un altro formalismo

Abbiamo sperimentato entrambe le soluzioni, descriviamo ora la prima e rimandiamo la presentazione della seconda alla sez. 6.

In primo luogo è stata definita una nuova regola che descrive la situazione in cui l'azione è proibita:

```
Insegnamento non_attribuibile_in_supplenza_a Docente :-
        Insegnamento e_uno_sdoppiamento,
        Numero esami_sostenuti_in Insegnamento,
        Numero < 250.
```

Quindi la sua conclusione negata è stata inserita nella parte condizionale della regola generale:

```
Insegnamento puo_essere_attribuito_in_supplenza_a Docente :-
    non_espletate_tornate_giudizi_di_idoneita,
    Insegnamento e_vacante,
    not (Insegnamento non_attribuibile_in_supplenza_a Docente).
```

• • •

Nella regola precedente è utilizzato l'operatore "not", questo operatore non coincide con la negazione logica ma esprime "negazione come fallimento" che, come Clark ha dimostrato [6], è consistente con la negazione logica sotto l'ipotesi della "closed world assumption".
Questo tipo di negazione è particolarmente utile in ambito legale poichè spesso i testi legislativi definiscono una norma generale seguita da una lista di eccezioni e, se una situazione non è "eccezionale", cioè non soddisfa le condizioni che definiscono l'eccezione, è naturale considerarla come una situazione ordinaria.
"Negazione come fallimento" non è però sempre adeguata per rappresentare la proibizione, poichè essa non è applicabile qualora non si conoscano tutte le eccezioni rilevanti o sia necessario utilizzare due tipi diversi di negazione [22].

Le altre regole del sistema specificano le altre forme di assegnamento e definiscono concetti richiamati all'interno delle regole.
Per rappresentare in maniera coordinata le modalità mediante le quali l'insegnamento può essere attribuito è stato necessario definire una regola ad alto livello che non corrisponde ad alcuna parte di testo.

```
Insegnamento puo_essere_attribuito_a Docente:
  Insegnamento puo_essere_attribuito_per_sostituzione_a Docente;
  Insegnamento puo_essere_attribuito_per_2_insegn_a Docente;
  Insegnamento puo_essere_attribuito_per_supplenza_a Docente.
```

La rappresentazione del testo della circolare interpretativa ha condotto alla ridefinizione della struttura di alcune delle norme precedenti ed alla definizione, per mezzo di nuove regole, di concetti vaghi delle leggi precedenti che erano stati rappresentati come condizioni elementari.
La formalizzazione completa della legge, composta da 55 regole, che rappresentano circa sette pagine di testo, è descritta in [23].
L'implementazione è stata realizzata in Prolog I su un personal computer.

5. UTILIZZO DELLA FORMALIZZAZIONE

La rappresentazione della normativa può essere utilizzata per fornire risposte a quesiti che riguardino l'applicazione della legge in specifiche circostanze.
Ad esempio, si può verificare se un insegnamento può essere attribuito ad un docente secondo una particolare modalità o lasciando al sistema il compito di determinare la modalità corretta e, data una data descrizione degli insegnamenti e dei docenti di una facoltà, possono essere richiesti quali insegnamenti e/o quali docenti soddisfino i requisiti richiesti dalla legge.

A questo scopo l'interprete Prolog viene utilizzato come meccanismo inferenziale che, in base alla descrizione della normativa ed un insieme di fatti che descrivono la situazione da indagare, effettua le deduzioni appropriate al fine di dimostrare l'ammissibilità di una determinata azione.

La base di conoscenza come è stata ottenuta per mezzo della traduzione dei testi di legge non è direttamente utilizzabile per diverse modalità di consultazione ma è necessario introdurre all'interno delle regole dei predicati di input/output e di controllo. Poichè l'introduzione di questi predicati distrugge la semantica dichiarativa della rappresentazione si è deciso di mantenere come rappresentazione della norma quella ottenuta dall'analisi dei testi legislativi e di far carico all'interprete di provvedere alle funzionalità di input/output e di controllo.
È stato sviluppato un interprete Prolog-like che incorpora una facility mediante la quale sia i fatti positivi che negativi sono registrati nel database e quando, nel corso della dimostrazione, l'interprete non trova nel database nè un fatto nè il suo contrario, interroga l'utente e trasforma la risposta ottenuta nel corrispondente fatto Prolog.

In relazione alla quantità di fatti inizialmente noti al sistema si hanno diversi tipi di utilizzo. In un primo tipo di utilizzo la base di conoscenza non contiene alcun fatto iniziale ed il sistema, per ciascuna condizione della regola alla quale la consultazione fa riferimento, richiede all'utente le caratteristiche della situazione da indagare, le risposte ottenute sono trasformate in fatti Prolog ed inseriti progressivamente nella base della conoscenza.
In un secondo tipo di utilizzo la base di conscenza contiene una descrizione dettagliata degli insegnamenti e dei docenti delle facoltà, queste informazioni saranno sfruttate per fornire una risposta ai quesiti richiesti senza ricorrere all'effettuazione di domande all'utente.
Un terzo genere di utilizzo prevede una base di conoscenza incompleta che non contiene tutte le informazioni richieste della legge. In questo caso, quando l'interprete nel corso della dimostrazione non troverà nella base di conoscenza il predicato corrispondente al tipo di requisito richiesto, la verifica sarà effettuata interagendo con l'utente con lo stesso meccanismo del caso 1.
Nel corso della consultazione il sistema, a richiesta, fornisce un tracciato dell'esecuzione con i frammenti del testo legislativo da cui le regole Prolog sono state ottenute.
Un esempio di consultazione è mostrato in fig. 1, nella figura l'input dell'utente è sottolineato ed i commenti sono racchiusi tra "**".

Il sistema di consultazione fa parte di un sistema di ufficio dedicato a supportare l'attività della segreteria didattica di Scienze della Informazione concernente l'attribuzione degli insegnamenti vacanti.
Le fasi dell'attività sono le seguenti:

a) dichiarazione della vacanza;
b) emissione del bando di concorso;
c) ricevimento delle domande;
d) verifica dei requisiti dei docenti che hanno presentato domanda;
e) conferimento della supplenza.

```
attribuzione-insegnamenti-vacanti.
Nome dell'insegnamento? analisi
Nome del docente? rossi
Quali modalità si intende utilizzare? 3
        1 - sostituzione
        2 - secondo insegnamento
        3 - supplenza
        4 - da determinare
È VERO CHE non_espletate_tornate_giudizi_idoneità? si
È VERO CHE analisi e_uno_sdoppiamento? si
PER QUALE X: X esami_sostenuti_in analisi? why
        SE analisi e_uno_sdoppiamento          E
           X è esami_sostenuti_in analisi      E
           X > 250
   ALLORA  analisi attribuibile_in_supplenza

            Art. 114, comma 1, 2 parte
  ...Non possono comunque essere coperti per supplenza
  gli insegnamenti sdoppiati salvo che il numero degli
  esami sostenuti negli insegnamenti stessi nell'ultimo
  anno accademico sia superiore a 250 per ciascun corso
  attivato.

PER QUALE X: X esami_sostenuti_in analisi? 300
        ...     ** verifica degli altri requisiti
                   dell'insegnamento **

È VERO CHE rossi e_un_ordinario? no
È VERO CHE rossi e_un_associato? si

                ** il sistema verifica
                   autonomamente che:
                   - rossi appartiene alla stessa
                     facoltà di analisi
                   - rossi insegna una materia
                     affine ad analisi **
CONCLUSIONE: L'insegnamento analisi può essere
             attribuito in supplenza al docente rossi
```

Fig. 1

Le varie fasi sono connesse con l'elaborazione ed il recupero di informazioni e comportano la produzione di un insieme di documenti.
Il sistema supporta ciascuna delle fasi precedenti offrendo delle facilities che permettono di registrare in un data base le informazioni riguardanti l'insegnamento vacante ed i docenti che hanno presentato domanda.
Questi dati saranno utilizzati successivamente dal sistema di consultazione nella fase d e nella produzione automatica dei moduli necessari nelle fasi a) b) c).

6. PROBLEMI DI RAPPRESENTAZIONE

Le clausole di Horn si sono rivelate un formalismo appropriato per la rappresentazione delle norme contenute in testi legislativi.
Per quanto riguarda le limitazioni imposte da questo tipo di logica: l'impossibilità di rappresentare conclusioni disgiuntive e negative, soltanto quest'ultima ha costituito un problema.
Infatti nel frammento di legge analizzato non sono state riscontrate regole che implicassero conclusioni disgiunte, i.e.

A or B se C

e si può ragionevolmente supporre che tali regole non ricorrano nei testi legislativi poichè la legge tende a specificare conclusioni definitive.

Per quanto riguarda il secondo aspetto per affrontare questo problema si è deciso di adottare, come criterio metodologico, un approccio di tipo sperimentale rivolto a modificare progressivamente il linguaggio di rappresentazione mediante la costruzione di nuovi intepreti.

Nella sez. 4 è stato mostrato come sia necessario trasformare una proibizione per l'impossibilità delle clausole di Horn, di rappresentare conclusioni negative. Per evitare tali trasformazioin è stata inclusa nella base della conoscenza la proibizione di raggiungere una conclusione.
In questa versione del sistema una conclusione di una regola può essere raggiunta se tutte le sue condizioni sono soddisfatte e non può essere provata alcuna conclusione contraria. Nel caso in cui nessuna conclusione negativa sia specificata il sistema si comporta come l'usuale interprete Prolog.
Questa soluzione ha il vantaggio che la formalizzazione risulta più simile alla struttura del testo originario ma ulteriori approfondimenti sono necessari al fine di comprendere le sue limitazioni e le analogie con il concetto di "negazione come inconsistenza" [10] e con altre forme di negazione avanzate recentemente [9].

Un altro problema che è stato riscontrato è che le definizioni di alcune regole fanno affidamento ad assunzioni implicite non specificate in alcun testo. Ad esempio una norma dell'art. 9 stabilisce che un insegnamento può essere attribuito in supplenza ad un professore ordinario senza specificare che il corso deve essere vacante e, analogamente, nell'art. 114 non è specificato se è necessario il consenso del docente o se l'attribuzione deve essere effettuata su sua richiesta.
Questi casi indicano come una base della conoscenza costituita unicamente dalla rappresentazione dei testi di legge non è sufficiente poichè conclusioni indesiderate possono essere successivamente derivate dalla formalizzazione.
Per permettere la rappresentazione di conoscenze implicite è stata fornita la possibilità di associare condizioni aggiuntive a ciascun predicato di una regola. Le condizioni aggiuntive agiscono come vincoli che devono essere soddisfatti precedentemente o successivamente alla verifica del predicato a cui sono associati.
La nostra esperienza ci ha permesso di verificare che, per disegnare programmi logici che descrivono una normativa in modo preciso ed affidabile, la formalizzazione dei testi legislativi deve essere arricchita da una descrizione delle entità a cui le norme fanno riferimento. Inoltre strutture per la rappresentazione a diversi livelli di astrazione e meccanismi di interazione tra i livelli sono necessari.
Attualmente stiamo analizzando questi problemi e stiamo sperimentando soluzioni rimanendo in un ambito logico.

7. CONCLUSIONI

Il dominio legale possiede particolari caratteristiche che lo differenziano da altri domini di conoscenza che sono stati analizzati e rappresentati, tra di queste di particolare importanza è il fatto che la conoscenza è espressa in testi scritti in linguaggio naturale.
Nell'articolo è stato presentato come delle norme legali possono essere formalizzate come programmi logici che, opportunamente arricchiti, costituiscono la base di conoscenza di un sistema esperto di tipo legale. In quest'ambito l'utilizzo di Prolog ha permesso uno stile di rappresentazione coinciso ed espressivo ed ha consentito uno sviluppo incrementale e per tentativi del sistema che si è dimostrato particolarmente adatto alla rappresentazione dei testi legislativi.

8. RINGRAZIAMENTI

Vorremmo ringraziare il prof. G. Degli Antoni per i suggerimenti forniti nel corso della ricerca e A. Masia e O. Zancolò per l'aiuto prestato nella comprensione della legge.

9. BIBLIOGRAFIA

[1] Allen L., SYMBOLIC LOGIC: A RAZOR EDGED TOOL FOR DRAFTING AND INTERPRETING LEGAL DOCUMENTS, Yale Law Journal, 1957

[2] Ashley K.D., REASONING BY ANALOGY: A SURVEY OF SELECTED AI RESEARCH WITH IMPLICATION FOR LEGAL EXPERT SYSTEMS, Proc. 1° Conference on Law and Technology, University of Houston, 1985

[3] Bertocchi R., RAPPRESENTAZIONE DI TESTI DI LEGGE IN PROLOG, 1° Convegno nazionale sulla programmazione logica, Genova 1986

[4] Ciampi C. (ed.), ARTIFICIAL INTELLIGENCE AND LEGAL INFORMATION SYSTEMS, North Holland, 1982

[5] Clark K.L., Mc Cabe F., PROLOG FOR EXPERT SYSTEMS in Machine Intelligence n. 10, Ellis Horwood, 1982

[6] Clark K.L., NEGATION AS FAILURE, in Logic and Data Bases, Gallaire H. (ed.), Plenum Press, New York, 1978

[7] Degli Antoni G., Zonta B., ANALYSIS OF LAW BY MEANS OF PETRI NETS: MOTIVATIONS AND METHODOLOGY, in Artificial Intelligence and Legal Information systems, Ciampi C. (ed.), North Holland, 1982

[8] Gardner A. v.d.L., THE DESIGN OF A LEGAL ANALYSIS PROGRAM, Proc. AAAI-83, Washington, 1983

[9] Gabbay D., Reyle U., N-PROLOG AN EXTENSION OF PROLOG WITH HYPOTHETICAL IMPLICATIONS, The Journal of logic programming, 1984, n. 3

[10] Gabbay D., Sergot M., NEGATION AS INCONSISTENCY, Research Report Imperial College, 1984, London

[11] Hammond P., Sergot M.J., A PROLOG SHELL FOR LOGIC BASED EXPERT SYSTEMS, Proc. 3rd BCS Expert Systems Conference, Cambridge, British Computer Society, 1983

[12] Hustler A., PROGRAMMING LAW IN LOGIC, Research Report CS-82-13, University of Waterloo, Canada, 1982

[13] PANEL ON ARTIFICIAL INTELLIGENCE AND LEGAL REASONING, Proc. IJCAI-85, Los Angeles 1985

[14] Marino A., Natali F. (eds.), 2° CONVEGNO INTERNAZIONALE SU LOGICA, INFORMATICA E DIRITTO, Firenze 1985

[15] McCarthy L.T., A COMPUTATIONAL THEORY OF LEGAL ARGUMENTS, LRP TR 13, Laboratory for Computer Science Research, Rutgers University, 1982

[16] Niblett R. (ed.), COMPUTER SCIENCE AND LAW, Cambridge University Press, 1979

[17] Pereira L.M., Oliveira E., PROLOG FOR EXPERT SYSTEMS: A CASE STUDY, Proc. IFAC, Leningrad, USSR 1983

[18] Reiter R., ON CLOSED WORLD DATABASES, in Logic and Data Bases, Gallaire H. (ed.), Plenun Press, New York, 1978

[19] Sanchez M., MODELLI ARITMETICI PER L'INFORMATICA GIURIDICA, Informatica e Diritto, Vol. 2, 1978

[20] Sergot M.J., PROSPECTS FOR REPRESENTING THE LAW AS LOGIC PROGRAMS, in Logic Programming, Academic Press, London, 1982

[21] Sergot M.J., REPRESENTING LEGISLATION AS LOGIC PROGRAMS, Research Report 1985, Imperial College, London

[22] Sergot M.J., Sadri F., Kowalski R.A., Kriwaczek F., Hammond P., Cory H.T., THE BRITISH NATIONALITY ACT AS A LOGIC PROGRAM, Research Report 1985, Imperial College, London

[23] Stagnoli M., SISTEMI DI CONSULTAZIONE BASATI SU LEGGI PER PROCEDURE UNIVERSITARIE, Tesi di Laurea in Scienze dell'Informazione, Anno Accademico 1985/86

[24] Waterman D.A., Peterson M.A., MODELS OF LEGAL DECISION MAKING, Report R-2717, Rand Corporation, Institute for Civil Justice, 1981

La ricerca in corso è stata patrocinata dal CNR nell'ambito del Progetto Strategico Sistemi Esperti.

UN'INTERFACCIA UOMO-MACCHINA INTERATTIVA PER OBJ

Vito Besana, Sergio Zocco
Dipartimento di Scienze dell'Informazione - Università di Milano

RIASSUNTO

Questo lavoro presenta una interfaccia interattiva per scrivere specifiche nel linguaggio OBJ.
L'interfaccia è basata su di un editor diretto dalla sintassi, richiamabile dall'interno dell'interprete OBJ, che aiuta il programmatore a costruire specifiche sintatticamente corrette, evidenziando in ogni momento le produzioni applicabili.
Inoltre, un apposito modulo verifica la sufficiente completezza della specifica.
L'interfaccia è stata implementata presso il Dipartipento di Scienze dell'Informazione dell'Università di Milano.

ABSTRACT

In this paper, and interactive interface for writing specifications in the OBJ language is presented.
The interface is based on a syntax directed editor which helps the programmer in constructing syntacically correct specifications by showing which productions can be applied at every point.
Furthermore, a completeness checker verifies the sufficient completeness of the specification.
The interface has been implemented at the Department of Computer Science of the University of Milan.

1. INTRODUZIONE

OBJ è un linguaggio di specifica, basato sulla logica equazionale, nato nell'ambito delle ricerche sull'ingegneria del software. Originariamente OBJ era stato pensato come linguaggio per la specifica di tipi di dati astratti, in seguito è stato sviluppato come linguaggio di specifica eseguibile, "general-purpose", adatto ad una notevole varietà di applicazioni.

Le continue evoluzioni che tale linguaggio sta subendo consentono oggi di vederlo anche come vero e proprio linguaggio di programmazione particolarmente adatto ad applicazioni di tipo matematico.

2. CARATTERISTICHE GENERALI DI OBJ

Un programma OBJ è costituito da uno o più oggetti opportunamente collegati tra di loro. Tali oggetti costituiscono le entità di base attraverso cui si definiscono le strutture di dati in OBJ. Lo schema seguente introduce la struttura sintattica di un oggetto.

‹nome oggetto›
‹lista delle sorte›
‹lista operatori›
‹lista variabili›
‹lista equazioni›

Ciascun oggetto ha un nome che deve essere univoco e può contenere una lista di sorte, ossia degli insiemi di dati, che vengono utilizzate per definire degli eventuali operatori; utilizzando poi delle variabili, opportunamente introdotte nell'oggetto, è possibile definire le proprietà di tali operatori attraverso delle equazioni: per tale motivo OBJ si dice basato sulla logica equazionale.

OBJ è inoltre un linguaggio eseguibile nel senso che è possibile testate rapidamente i programmi specificati attraverso la semplice esecuzione di test ad alto livello. L'eseguibilità delle specifiche OBJ si basa sull'interpretazione delle equazioni come regole di riscrittura. In tal modo, definito un certo oggetto con delle equazioni, e data una certa espressione da ridurre, una delle equazioni, vista come regola di riscrittura, può essere applicata a tale espressione, se quest'ultima contiene un'istanza del lato sinistro del-

l'equazione in questione. È così possibile ridurre l'espressione di partenza riscrivendo il lato sinistro dell'equazione in essa contenuta tramite il corrispondente lato destro.

Questo schema di riscrittura procede fino a quando all'espressione che via via si ottiene non può più essere applicata nessuna regola di riscrittura.

L'eseguibilità delle specifiche OBJ è quindi definita attraverso un algoritmo di riscrittura che si occupa di trovare gli eventuali "match" fra espressioni da ridurre ed equazioni e in seguito esegue la riscrittura vera e propria attraverso il meccanismo precedentemente introdotto.
Dato che non è sempre facile definire le proprietà degli operatori attraverso equazioni e viste le difficoltà che si incontrano nella stesura di specifiche OBJ, è stato realizzato un editor guidato da sintassi che facilita questo compito. In particolare in questo lavoro si presentano le caratteristiche principali di tale strumento che consente di semplificare notevolmente le operazioni di scrittura di programmi OBJ, garantendo tra l'altro controlli sulla sufficiente completezza degli stessi.

L'editor guidato da sintassi è stato realizzato per il linguaggio OBJ-T per il quale esiste un interprete presso l'Università di Milano dove già da alcuni anni OBJ è stato oggetto di studi e ricerche.

3. PRESENTAZIONE DELL'INTERFACCIA

Un utente dell'interprete OBJ che desideri scrivere un oggetto "colloquiando" con il sistema, può digitare il comando SDEOBJ chiamando l'editor guidato da sintassi.

Lo schermo gli si presenta diviso in tre parti (vedi fig. 1): le prime 7 righe vengono utilizzate per colloquiare con l'utente; nell'ottava riga vengono visualizzati i messaggi di errore e i commenti del sistema; la restante parte dello schermo è occupata dal "contesto", cioè l'oggetto che viene espanso a seconda delle scelte e delle dichiarazioni dell'utente.

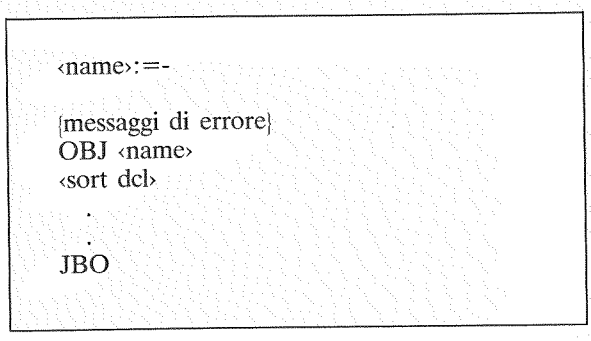


Fig. 1

Vediamo ora in dettaglio come avviene l'interazione con l'utente.

4. DICHIARAZIONE DEL NOME E DELLE SORTE

Inizialmente, nella parte di video rappresentante il contesto, apparirà la struttura di un oggetto così fatta:

OBJ ‹NAME DCL›

‹SORT DCL›
‹OPS DCL›
‹EQNS DCL›
JBO

mentre nella prima parte dello schermo apparirà la scritta:

‹NAME DCL›::= ‹NAME›/‹PREV OBJ NAMES› (1/2)

Ciò che è scritto tra parentesi angolari è quello che dovrà essere espanso. Ad esempio "‹NAME DCL›" (dichiarazione del nome) può essere espanso in una delle due alternative proposte sullo schermo (‹NAME› oppure ‹NAME›/‹PREV OBJ NAMES› cioè nome seguito da uno "/" e da una eventuale lista di nomi di oggetti ereditati).

Se l'utente batte subito il tasto ‹CR›, gli viene chiesto se vuole definire un oggetto. Se invece batte un carattere diverso da 1 o 2 compare il messaggio di errore "La scelta è fra 1 e 2" e vengono ripetute le due alternative.

A seconda della scelta fatta dall'utente comparità nel contesto al posto di "‹NAME DCL›" "‹NAME›" o "‹NAME›/‹PREV OBJ NAMES›".

Nella parte dello schermo relativo al menù comparità a questo punto la richiesta del nome dell'oggetto.
‹NAME›:=-
Se si batte ‹CR› viene chiesto se si vuole definire un oggetto, altrimenti viene controllato se il nome non appartenga ad un oggetto già caricato nel sistema. Se ad esempio l'utente batte il nome "lista" dopo aver scelto la seconda alternativa propostagli dal primo menù, il contesto si presenterà in questo modo:

OBJ lista / ‹PREV OBJ NAMES›
‹SORT DCL›
.
.
.
JBO

Ora viene la dichiarazione degli oggetti che "lista" eredita. All'utente viene chiesto ripetutamente il

nome dell'oggetto.

‹NAME›:=-

ed ogni volta l'editor controlla se effettivamente l'oggetto è già stato inserito nel sistema; se supera la verifica viene aggiunto nel contesto. Il procedimento si ripete fino a che l'utente batte solo il tasto ‹CR› per indicare che non vuole più dichiarare altri oggetti ereditati.

La dichiarazione delle sorte avviene in modo analogo a quella dei nomi degli oggetti. Inizialmente viene presentato il menù:

‹SORT DCL›::= SORTS ‹CURRSORTS› (1)
SORTS ‹CURRSORTS› / ‹PREV-SORTS› (2) ?-

Se ad esempio si vogliono dichiarare solo 2 sorte ereditate s1 e s2 è necessario scegliere il secondo caso. Quando compare "‹SORT›:=" si deve battere ‹CR› per indicare che non si vogliono dichiarare nuove sorte nell'oggetto. Nel contesto verrà cancellato "‹CURRSORTS›" e verrà riproposto "‹SORT›:=" come espansione di "‹PREVSORTS›".
Dopo aver battuto s1 l'editor controlla se effettivamente la sorta appartenga ad un oggetto ereditato (cioè dichiarato dopo il simbolo "/"). L'oggetto nel contesto ora si presenta così:

OBJ example
SORT / s1 ‹SORT›
‹OPS DCL›
.
.
.
JBO

Dopo aver battuto anche la sorta s2, l'editor ripropone la dichiarazione di un'altra sorta (compare "‹SORT›:=" all'inizio dello schermo). Dando ‹CR› si passa automaticamente a trattare gli operatori.
Nel caso la sorta non sia corretta (cioè un nome ripetuto o una sorta ereditata non esistente) viene dato un messaggio di errore e viene riproposta la richiesta di una sorta.

5. DICHIARAZIONE DEGLI OPERATORI

Terminata la dichiarazione delle sorte l'editor espande automaticamente "‹OPS DCL›" nelle tre componenti:

"‹OK OPS DCL›", "‹ERR OPS DCL›", "‹FIX OPS DCL›" che verranno trattate una alla volta nell'ordine dato.

Inizialmente viene chiesto all'utente se vuole definire operazioni di quel tipo:

‹OK OPS DCL›::= OK-OPS ‹OPSLIST› | & (1/2)-
dove il simbolo & sta per "niente".

Se la risposta è negativa si passa al tipo di operazioni successive, altrimenti nel contesto viene espanso "‹OK OPS DCL›" con la parola chiave OK-OPS seguita da "‹OPSLIST›".
Nel menù che può assumere un operatore:

‹OPKIND›:= ‹PREFIX OPS DCL› (1)
‹PREFIXPAR OPS DCL› (2)
‹POSTFIX OPS DCL› (3)
‹INFIX OPS DCL› (4)
‹CONST OPS DCL› (5)
‹COERCION DCL› (6)
‹DISTR OPS DCL› (7) ?-

l'utente batterà un numero fra 1 e 7 a seconda del tipo di operatore che vuole definire. Nel caso battesse un carattere non previsto, dopo la visualizzazione di un messaggio di errore, verrebbe ripresentato il menù precedente (si ha lo stesso comportamento ogniqualvolta viene presentata una lista di alternative fra cui scegliere).

Supponiamo che l'utente voglia definire questi due operatori:

op1 : s1 s2 s1⟶ s2
_op2_ : s1 s1 ⟶ s1

Il primo è di tipo prefisso parentesizzato mentre il secondo è di tipo infisso.

Per inserire il primo operatore l'utente deve scegliere nel menù propostogli "‹PREFIXPAR OPS DCL›" battendo il numero relativo (2). A questo punto nel contesto apparirà la struttura di un operatore di questo genere.

‹OPNAME› : ‹SORTLIST› ⟶ ‹SORT› ‹ATTR›
e nel menù:
‹OPNAME›:=_

l'editor è in attesa che l'utente batta il nome dell'operatore. Dopo che l'utente ha dato il nome l'editor chiede la prima sorta del dominio. Dopo aver ricevuto s1 come input dall'utente il contesto si presenta così

op1 : s1 ‹SORT› ⟶ ‹SORT› ‹ATTR›

L'editor continua ad aspettarsi nuove sorte del dominio fino a che non gli si batte il tasto di ‹CR› dopo "‹SORT›:=". Ora viene chiesta la sorta del codominio.
Per ogni sorta viene controllato che questa sia definita all'interno dell'oggetto stesso o appartenga ad oggetti ereditati.

Infine si passa all'immissione degli attributi. Se l'u-

tente sceglie di dichiararne, l'editor gli propone di selezionarli fra una lista che dipende dal tipo di operatore dichiarato. Lo schema seccondo cui gli attributi vengono proposti è questo:

- hidden e permuting per qualsiasi tipo di operatore (tranne le coercion che non hanno attributi).
- commutative e associative per operatori infissi binari il cui codominio sia uguale alle due sorte del dominio.
- equality per operatori binari con le due sorte del dominio uguali e con codominio booleano.
- livello di precedenza per operatori diversi da costanti e operatori parentesizzati, con codominio uguale alla prima e ultima sorta del dominio.

Nel caso dell'operatore op1 dell'esempio gli unici attributi possibili sono hidden e permuting, mentre per _op2_ è possibile scegliere anche commutative e associative.

La dichiarazione di operatori error e fix avviene allo stesso modo di quelli ok. L'unica differenza è che fra gli operatori error e fix non vi possono essere coercions.

6. LA DICHIARAZIONE DELLE VARIABILI

Prima di scrivere le equazioni dell'oggetto devono essere dichiarate le variabili che poi verranno utilizzate nei termini delle equazioni.

Dopo che l'utente ha risposto affermativamente alla solita domanda "vuoi definire delle variabili?" ("‹VARS DCL›::= VARS ‹VARLIST› | & (1/2)"), nel menù compare:

"‹VAR›:=".

Supponiamo di voler dichiarare le seguenti variabili: var1, var2 di sorta s1 e var3 di sorta s2. Scrivendo "var1" seguito da ‹CR› nel contesto avremo:

```
.
.
VARS var 1 ‹VAR› : ‹SORT›
.
.
```

(l'editor controlla che la variabile var1 non sia già stata dichiarata). Il sistema ora chiede un'altra variabile ("‹VAR›:=") e noi battiamo "var2". Alla terza richiesta, battendo il tasto ‹CR› viene chiesta la sorta delle variabili ("‹SORT›:="), mentre nel contesto avremo:

```
.
.
VARS var1 var2 : ‹SORT›
.
.
```

Se battiamo ‹CR› l'editor cancella i due nomi delle variabili var1 e var2 e ripropone una nuova dichiarazione; se invece digitiamo una sorta non esistente, l'editor invia un opportuno messaggio di errore e richiede la sorta. Digitando "s1", supponendo che sia una sorta corretta, nel contesto avremo:

```
.
.
VARS var1 var2 : s1
       ‹VARLIST›
‹EQNS DCL›
JBO
```

Nello stesso modo in cui sono state definite var1 e var2 si definisce var3 con sorta s2.

L'editor a questo punto propone l'inserimento di altre variabili; battendo ‹CR› si passa invece a trattare le equazioni. "‹EQNS DCL› verrà espanso automaticamente in "‹OK EQNS›" e "‹ERR EQNS›". Vediamo ora come si presenta l'oggetto:

```
.
.
VARS var1 var2 : s1
       var3 : s2
‹OK EQNS DCL›
‹ERR EQNS DCL›
JBO
```

7. DICHIARAZIONE DELLE EQUAZIONI

La differenza fra equazioni di tipo ok ed equazioni d'errore è questa: le prime sono uguaglianze di termini che non danno valori di errore; le seconde invece devono avere il termine destro manifestamente in errore. Quindi la scrittura delle equazioni ok ed err si svolge nello stesso modo; è l'editor che, a seconda del tipo dell'equazione, effettua certi controlli piuttosto che altri.
Vi sono due tipi di equazioni: equazioni semplici e condizionali. Il sistema propone la scelta prima di iniziare il dialogo interattivo per la scrittura di una equazione:

```
‹EQN›:= (‹EXP› = ‹TERM›) | (‹EXP› = ‹TERM›
        ‹COND›) (1/2)
```

Se si sceglie la seconda alternativa, "‹COND›" viene espanso in "IF ‹TERM›" oppure in "IF NOT ‹TERM›". In entrambi i casi l'operatore più esterno del termine indicato deve avere come codominio la sorta bool.

Vediamo ora come avviene la scrittura delle equazioni attraverso un esempio significativo. Supponiamo di voler scrivere la seguente equazione ok:
(constant oper var1 distr = a(var1))
avendo dichiarato in precedenza gli operatori:

a : s1 ⟶ s2
constant : ⟶ s3
_oper_distr : s3 s1 ⟶ s2
e la variabile var1 di sorta s1.
Dopo aver scelto il caso di una equazione non condizionale, questa di presenta nella forma:

** (‹EXP› = ‹TERM›)

(con due asterischi sarà indicata l'equazione in via di espansione, mentre con una freccia, ciò che deve essere digitato).
Il motivo per cui nel lato sinistro viene richiesta una espressione invece che un termine è questo: la sintassi prevede che un temine possa essere sia una espressione che una variabile, e quindi il sistema, richiedendo che il lato sinistro sia necessariamente maggiore o uguale al destro rispetto alla profondità, forza l'utente a scegliere come termine una espressione; tale vincolo non è mantenuto per il lato destro.

‹EXP›:= ‹PREFIX EXP› (1)
‹PREFIXPAR EXP› (2)
‹POSTFIX EXP› (3)
‹INFIX EXP› (4)
‹CONST EXP› (5)
‹DISTR EXP› (6) ? ⇒ 6

‹OPNAME›:= ⇒ _oper_distr

Ora l'editor, dopo aver controllato l'esistenza di questo operatore, analizza la stringa alla ricerca dei place-holder che sostituirà nel contesto con il non-terminale "‹TERM›".

** (‹TERM› oper ‹TERM› distr = ‹TERM›)

ora viene espanso il primo temine:

‹TERM›:= ‹VAR› | ‹EXP› (1/2) ⇒ 2

‹EXP›:= ‹PREFIX EXP› (1)
‹PREFIXPAR EXP› (2)
‹POSTFIX EXP› (3)
‹INFIX EXP› (4)
‹CONST EXP› (5)
‹DISTR EXP› (6) ? ⇒ 5

‹OPNAME›:= ⇒ constant

L'editor controlla l'esistenza di questo operatore e la consistenza fra la sorta della costante e la prima sorta del dominio di "_oper_distr". In caso questa mancasse, dopo aver dato un opportuno messaggio di errore, verrebbe riproposto: "‹OPNAME›:="

** (constant oper ‹TERM› distr = ‹TERM›)

‹TERM›:= ‹VAR› | ‹EXP› (1/2) ⇒ 1

‹VAR›:= ⇒ var1

Ora inizia l'espansione del lato destro.

‹TERM›:= ‹VAR› | ‹EXP› (1/2) ⇒ 2

‹EXP›:= ‹PREFIX EXP› (1)
‹PREFIXPAR EXP› (2)
‹POSTFIX EXP› (3)
‹INFIX EXP› (4)
‹CONST EXP› (5)
‹DISTR EXP› (6) ? ⇒ 5

‹OPNAME›:= ⇒ a

Viene controllata la consistenza fra lato sinistro e destro (codominio di "a" uguale al codominio di "_oper_distr"). Dopo aver visto che l'operatore è monodimensionale:

** (constant oper va1 distr = a (‹term›))

‹TERM›:= ‹VAR› | ‹EXP› (1/2) ⇒ 1
‹VAR›:= ⇒ var1

** (constant oper var1 distr = a (var1))
‹EQN›
e viene proposta la struttura di un'altra equazione.

Dopo le equazioni di errore, l'oggetto è terminato e, come detto precedentemente, si chiede se l'oggetto deve essere inserito nel sistema. In caso affermativo, si può poi chiedere all'interprete OBJ di immaginare l'oggetto appena inserito o di eseguire direttamente delle "RUN" per testare la correttezza della specifica.

8. LA VERIFICA DELLA SUFFICIENTE COMPLETEZZA NELL'EDITOR GUIDATO DA SINTASSI

L'interfaccia è stata disegnata in modo tale che l'utente non possa commettere errori sintattici nella scrittura delle specifiche.
Inoltre l'editor effettua controlli su tutti quegli errori semantici che sono trattabili staticamente.

La teoria della sufficiente completezza, studiata da Guttag e Hornig dà dei criteri per verificare la correttezza delle specifiche a livello semantico, che sono stati applicati alla scrittura degli oggetti OBJ.
Vediamo ora quali sono le funzionalità aggiunte all'editor per trattare la verifica della sufficiente completezza.

9. IL PARTIZIONAMENTO DEGLI OPERATORI

Perchè si possa implementare la verifica della sufficiente completezza, è necessario dividere gli operatori dell'oggetto in tre insiemi.

L'insieme 0 è formato da quegli operatori la cui sorta del codominio non appartiene a "currsorts", cioè all'insieme delle sorte definite nell'oggetto e non ereditate.

L'insieme C è formato da quegli operatori con codominio di sorta appartenente a "currsorts" e con dominio o monodimensionale e di sorta ereditata o pluridimensionale.

L'insieme E è formato dagli operatori con dominio monodimensionale con sorta in "currsorts" e con codominio anch'esso in "currsorts".

L'appartenenza degli operatori ad uno di questi insiemi viene stabilita al termine della dichiarazione degli attributi. Ad esempio se vogliamo scrivere la specifica di una coda circolare,

OBJ circular-queue / int
SORTS queue / int
OK-OPS
read : queue ⟶ int
enqueue : int queue ⟶ queue
dequeue : queue ⟶ queue
init: ⟶ queue
circulate : queue ⟶ queue
.
.
.

al termine della scrittura di ogni operatore, viene calcolato l'insieme al quale appartiene.
Vediamo come vengono partizionati gli operatori.

read : queue ⟶ int	O
enqueue : int queue ⟶ queue	C
dequeue : queue ⟶ queue	E
init : ⟶ queue	C
circulate : queue ⟶ queue	E

Al termine della dichiarazione degli operatori, il sistema propone un insieme di lati sinistri di equazioni tratti dagli insiemi OTERMS ed ETERMS come dettato dalla teoria della sufficiente completezza. L'utente ha così una indicazione sulla forma che le equazioni dell'oggetto dovranno avere.

Nel caso della coda circolare, l'editor, al termine della dichiarazione dell'operatore "circulate", stampa nella zona dello schermo utilizzato per i menù i modelli per i lati sinistri delle equazioni:

read (init)
read (enqueue (X~, Y~))
dequeue (init)
dequeue (enqueue (X~, Y~))
circulate (init)
circulate (enqueue (X~, Y~))

dove "X" sta per una sorta del dominio appartenente a "currsorts", "Y" altrimenti mentre "~" indica una sequenza di operatori.

10. I CONTROLLI SULLA COMPLETEZZA

Giunti alla scrittura delle equazioni, il sistema controlla il lato sinistro di ogni equazione. In particolare, dopo la scelta dell'operatore con livello di nesting uguale a due, l'editor è in grado di sapere la forma a cui ricondurre l'equazione. Se, ad esempio, sempre nel caso della coda circolare, l'utente è arrivato a questo punto dell'espansione di una equazione:

(read (enqueque (‹TERM›, ‹TERM›)) = ‹TERM›)

il sistema riconosce che l'equazione è della forma;

O(C(X,Y~),W~)) = z

dove read è l'operatore O ed enqueue è l'operatore C. Quindi in questo caso il sistema attenderà il termine della scrittura dell'equazione per verificare se è monotona rispetto alla funzione di nesting. Se non lo è, l'editor invia un opportuno messaggio all'utente; altrimenti aspetta la scrittura di tutte le restanti equazioni per verificare se esiste una sequenza di riscrittura come stabilito dalla teoria della sufficiente completezza.

Supponiamo ora che l'utente stia dichiarando una equazione di questo tipo:

(dequeue (enqueue (‹TERM›, ‹TERM›)) = ‹TERM›)

In questo caso l'editor riconosce che l'equazione ha la forma:

E(C(X,Y~),W~)) = z

Ora il sistema attende la scrittura completa dell'equazione per verificare la convertibilità. Viene cioè richiamato un algoritmo che controlla che sia rispettata la relativa condizione.

Nel caso non sia rispettata la condizione l'editor invia il messaggio: "L'equazione non rispetta la condizione di convertibilità".

Se un'equazione si presenta in una forma diversa da quella richiesta dalla procedura di verifica, il sistema informa l'utente che la correttezza di quella equazione non può essere testata (appare il messaggio: "L'equazione non è verificabile"). Ciò non significa, tuttavia, che l'equazione sia necessariamente errata.

11. RINGRAZIAMENTI

Si ringraziano J.A. Goguen per aver messo a dispo-

sizione il listato originale dell'interprete implementare presso UCLA, il Prof. G. Mauri, il Prof. G. Degli Antoni e il Dott. E. Ciapessoni per la proficua collaborazione e per l'assistenza fornita durante l'intero svolgimento del lavoro, il Dott. G. Sabino e il Dott. L. Fusi che con le loro precedenti ricerche hanno gettato le basi di questo lavoro.
Il lavoro è stato svolto nell'ambito del CP Project dell'Università di Milano e della Honeywell Information Systems Italia.

12. BIBLIOGRAFIA

[1] M. Bidoit, C. Choppy, S. Kaplan, UN ENVIRONMENT DE SPECIFICATION ALGEBRIQUE, Rapport de Recherche n. .144, 1983

[2] M. Bidoit, M.C. Gaudel, SPECIFICATION DES CAS D'EXCEPTIONS DANS LES TYPES ABSTRAITS ALGEBRIQUES: PROBLEMES ET PERSPECTIVES, Université de Paris-Sud, Route de Nozay, 1983

[3] L. Fusi, ASPETTI TEORICI DEL LINGUAGGIO OBJ E SUA IMPLEMETAZIONE IN LINGUAGGIO LISP, Università degli studi di Milano, 1984

[4] K. Futatsugi, J.A. Goguen, J. Jouannaud, J. Meseguer, PRINCIPLES OF OBJ2, SRI International, Stanford University, 1985

[5] H. Glaser, C. Hankin, D. Till, PRINCIPLES OF FUNCTIONAL PROGRAMMING, Prentice/Hall, 1984

[6] J.A. Goguen, J. Meseguer, OBJ-1, A STUDY IN EXECUTABLE ALGEBRAIC FORMAL SPECIFICATION, SRI International, 1981

[7] J.V. Guttag, J.J. Horning, THE ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES, Acta Informatica 10, 1978

[8] G. Mauri, OBJ, UN LINGUAGGIO DI PROGRAMMAZIONE BASATO SULLA LOGICA EQUAZIONALE, Note di Software 21/22, 1983

[9] G. Sabino, PROGETTO E ARCHITETTURA DI UNA INTERFACCIA UTENTE PER OBJ, Università degli studi di Milano, 1984

[10] J.J. Tardo, THE DESIGN, SPECIFICATION, AND IMPLEMENTATION OF OBJ-T: A LANGUAGE FOR WRITING AND TESTING ABSTRACT ALGEBRAIC PROGRAM SPECIFICATIONS, Phd. thesis, UCLA,1981

SISTEMI ESPERTI E RAPPRESENTAZIONE ETEROGENEA DELLA CONOSCENZA IN MEDICINA: L'ESEMPIO DELL'URGENZA TOSSICOLOGICA

Stefania Bandini
Dipartimento di Scienze dell'Informazione - Università di Milano

Luca Spampinato
Quinary s.r.l. - Milano

RIASSUNTO

In questo lavoro vengono discussi alcuni problemi riguardanti la rappresentazione eterogenea di conoscenza esperta nella costruzione di sistemi esperti nell'ambito della medicina.
Dopo l'analisi delle conoscenze impiegate dal medico specialista nella diagnosi e nel trattamento di intossicazioni da funghi velenosi a lunga incubazione, vengono descritti i formalismi di rappresentazione scelti.

ABSTRACT

The problem of heterogeneous knowledge representation in medicine is discussed in this paper.
After the analysis of expert knowledge employed in the diagnosis and treatment of long incubation mushroom poisoning, the representation formalism choices are illustred.

1. INTRODUZIONE

I sistemi esperti hanno trovato nella medicina un campo molto fecondo sia per quel che riguarda le applicazioni, che come settore sul quale sperimentare scelte e direzioni di ricerca per la crescita stessa delle tematiche che più concernono i problemi della rappresentazione della conoscenza nell'Intelligenza Artificiale.
Un numero abbastanza alto di esperti; i problemi specialistici circoscritti ma spesso non del tutto formalizzati; struttura ben definita della disciplina stessa: questi non sono che alcuni tra i fondamentali motivi che hanno portato la ricerca sui sistemi esperti a contatto con la medicina. Non è comunque trascurabile l'esigenza della medicina stessa di trovare nei sistemi di supporto automatici un valido strumento che può coinvolgere tutti i settori dell'operare medico e di cui i sistemi esperti non sono che un esempio.

I primi successi ottenuti nella progettazione e nella realizzazione di sistemi esperti per il settore della medicina non hanno comunque risolto la grande quantità di problemi che sorgono in questa direzione: una delle questioni fondamentali che sono ancora aperte riguarda la rappresentazione della conoscenza medica. I sistemi esperti basati su regole [1] non sono che uno degli esempi di metodologie di rappresentazione impiegati, mentre una continua collaborazione tra progettisti di sistemi basati sulla conoscenza e medici ha ormai messo in risalto la necessità di indagare quali formalismi di rappresentazione sono più idonei per permettere la costruzione della base della conoscenza il più possibile strutturata in modo da contenere conoscenze tra loro diverse e rappresentate senza tradire la natura della conoscenza stessa coinvolta.

In questo lavoro viene presentato il progetto di un sistema esperto in fase di prototipizzazione presso il Dipartimento di Scienze dell'Informazione e realizzato in collaborazione con la Cattedra di Medicina d'Urgenza del Policlinico di Milano; esso utilizza rappresentazioni eterogenee della conoscenza impiegata dal medico in uno specifico settore.

L'ambito di competenza a cui si è rivolta la nostra attenzione concerne l'emergenza tossicologica e, in particolare, la diagnosi e il trattamento di intossicazioni dovute all'ingestione di funghi velenosi a lunga incubazione.

Dopo una prima fase di descrizione dell'ambito medico, verranno analizzati i vari tipi di conoscenze impiegate dall'esperto nel trattamento delle intossicazioni da funghi velenosi a lunga incubazione e verrà fornita una descrizione dei formalismi di rappresentazione adottati per la prototipizzazione del sistema esperto.

2. L'AMBITO APPLICATIVO

Il compito riservato al reparto di pronto soccorso è quello di presentare le cure immediate nei casi di emergenza. Tale compito copre una vastissima gamma di situazioni e per poter affrontare casi urgenti tra loro differenti gli operatori del reparto spesso ricorrono al consulto degli specialisti.
Possiamo schematicamente individuare alcune caratteristiche fondamentali dell'attività di pronto soccorso:

- il suo servizio è attivo 24 ore su 24, cioè anche in fasce orarie in cui la reperibilità del personale specialistico diminuisce, a scapito dell'efficienza del reparto stesso la quale dipende anche dalla possibilità di trovare il giusto specialista nel giusto momento. Vi sono poi casi in cui la difficile reperibilità dell'assistenza specialistica è dovuta alla rarità di determinate classi di competenza;

- la diagnosi, o, almeno, le ipotesi diagnostiche devono essere effettuate con la massima urgenza al fine di poter avviare i più idonei interventi: nei casi d'emergenza questo compito è reso difficoltoso dal fatto che i dati necessari alla formulazione della diagnosi possono essere incompleti, affetti da disturbi, o, addirittura, mancare;

- anche in mancanza di una diagnosi di certezza devono comunque venire attivati degli interventi: di solito si tratta di interventi generali rispetto alla classe di patologie a cui il caso può appartenere, ma indispensabili alla sopravvivenza del paziente, nell'attesa che dati più certi permettano al medico di formulare la diagnosi esatta e prendere decisioni ad hoc.

Nel caso specifico dell'emergenza tossicologica, un'assistenza non immediata ed appropriata può risultare fatale per il paziente, mentre le competenze sugli avvelenamenti e sul loro trattamento sono altamente specialistiche e le ricerche in questo settore hanno una evoluzione continua.
Alcuni aspetti caratteristici dell'emergenza tossicologica possono essere individuati secondo questi punti:

- Un numero abbastanza alto di agenti tossici tra loro differenti presenta una sintomatologia comune o molto simile, specialmente nei primi stadi dell'avvelenamento: questo fatto rende difficile la precisa identificazione dell'agente che ha causato l'intossicazione e mette il medico in condizioni di incertezza riguardo all'instaurazione delle terapie.

- Un punto cruciale dell'emergenza tossicologica consiste nella formulazione della diagnosi di fase: come per un largo insieme di patologie, anche molte forme di intossicazione sono caratterizzate da un decorso clinico che può essere schematicamente suddiviso in una sequenza di fasi, dove ogni fase è evidenziata da una particolare sintomatologia che la caratterizza.
La conoscenza esplicita della sequenza delle fasi da parte del medico è quindi un requisito fondamentale per l'intervento urgente nelle intossicazioni e, parallelamente per il controllo del decorso clinico.

- Un'alta pecentuale di forme di intossicazione richiede interventi tempestivi di tipo generale, il cui scopo è rivolto principalmente a restaurare l'omeostasi del paziente.

- Le condizioni del paziente devono essere tenute in costante ed assidua osservazione; il riconoscimento delle fasi in evoluzione e le decisioni riguardanti le terapie più opportune da intraprendere richiedono operazioni di continuo monitoraggio su tutto il complesso dei dati del paziente (segni, sintomi e risultati degli esami di laboratorio).

2.1 Le intossicazioni da funghi velenosi a lunga incubazione

Nel nostro lavoro il problema dell'emergenza tossicologica è stato circoscritto ai funghi velenosi a lunga incubazione.
Per la scelta di questo specifico tema sono stati determinati fattori quali la presenza costante dell'esperto nel gruppo di lavoro, una buona letteratura sulla materia, una casistica ben documentata e metodologie di trattamento ormai consolidate.
L'alta pericolosità di questi funghi è dovuta alla loro elevata tossicità, al lungo periodo di tempo (6/18 ore) che intercorre tra l'ingestione e la comparsa di una sintomatologia e al fatto che non esistono antidoti.

Le intossicazioni da funghi a lunga incubazione possono essere ricondotte a tre diverse sindromi [8]:

- falloidea
- orellanica
- giromitrica

Fra tali sindromi sono compresi gli avvelenamenti più pericolosi per l'uomo (falloidee e parafalloidee) con sintomatologia prevalentemente gastroenterica e talvolta (orellanica) con complicanze o elettività di azio-

ne a livello renale. La sintomatologia iniziale presenta analogie molto forti per tutte e tre le sindromi, ma l'identificazione esatta del fungo che ha causato l'intossicazione diventa cruciale sia per poter seguire il decorso clinico che per le condotte terapeutiche da instaurare.

Per determinare con precisione l'dentità del fungo esistono solo pochi mezzi:

- il paziente può fornire una esatta descrizione del fungo esplicitandone le caratteristiche o lo può riconoscere attraverso immagini fotografiche che gli vengono sottoposte;
- l'analisi micologica dei residui di fungo che il paziente può recare con sè o che sono stati recuperati nei materiali di deiezione (vomito o feci) permette una esatta identificazione.

In realtà spesso capita che il paziente non sia in grado di fornire una esatta descrizione del fungo ingerito (ad esempio, i bambini o chi ha assunto funghi cucinati) e i centri sanitari in cui è operante un laboratorio che pratica analisi micologiche sono assai rari e i tempi in cui possono essere portate a termine tali analisi sono superiori a quelli entro i quali è necessario prendere delle decisioni; tutti questi fattori fanno sì che la reale possibilità di riconoscimento del fungo sia affidato alla personale esperienza del medico, così come ad essa è dato il compito di stabilire la diagnosi di fase e di instaurare le terapie ad hoc.

L'intossicazione da funghi velenosi a lunga incubazione è un tipico caso di avvelenamento con decorso clinico caratterizzato da una sequenza di fasi [7] (tab. 1, tab. 2, tab. 3).

La capacità di individuare anche da vaghe descrizioni e dalla sintomatologia l'identità del fungo e quella di formulare una diagnosi di fase sono caratteristiche non sempre possedute da ogni medico e la reperibilità di specialisti del settore generalmente è scarsa.

AMANITA PHALLOIDES
• fase di latenza
• fase coleriforme
• fase dell'epatite anitterica
• fase della caduta dei fattori di coagulazione o della insufficienza epatica
• fase della discesa enzimatica
• fase della risoluzione oppure del coma epatico e del seguente exitus.

Tab. 1

GIROMITRA
• fase di latenza
• fase neurogastroenterica
• fase neuroepatica
• fase della restitutio ad integrum o del no-back point

Tab. 2

CORTINARIUS ORELLANUS
• fase di latenza
• fase dei disturbi a carico dell'apparato gastroenterico
• fase dell'insufficienza renale e della nefropatia tubulo-interstiziale

Tab. 3

2.2 Un sistema esperto per le intossicazioni

Tutte queste considerazioni permettono di vedere come il settore dell'emergenza tossicologica e in particolare i problemi legati all'avvelenamento da funghi velenosi a lunga incubazione può essere un fertile campo d'applicazione per i sistemi automatici di supporto ai medici. Ci troviamo infatti di fronte a problemi che posseggono requisiti sufficienti per potenziali applicazioni di sistemi esperti [1]:

- la conoscenza e i dati riguardanti il dominio sono in parte reperibili in forma di letteratura e fattualmente documentabili;
- molte conoscenze sono basate su esperienze accumulate negli anni da un certo numero di specialisti del settore;
- il dominio riguarda una serie di prestazioni costose in termini di rischio e tali da giustificare l'impresa dello sviluppo di un sistema esperto;
- il problema è logicamente divisibile in passi distinti e presenta una alta modularità per la sua rappresentazione;
- la conoscenza di cui deve essere dotato il sistema

esperto ha carattere sostanziale e appartiene ad un numero ristretto di specialisti.

Il lavoro qui intrapreso è infatti volto alla realizzazione di un sistema esperto in grado di assistere il medico di pronto soccorso nel trattamento di pazienti colpiti da intossicazione da funghi velenosi a lunga incubazione; scopo applicativo è quello di garantire una ampia diffusione delle competenze necessarie per trattare questo tipo di casi.

3. DAL PROBLEMA ALLE CONOSCENZE COINVOLTE

Dalle ormai numerose esperienze nel campo delle applicazioni dell'Intelligenza Artificiale risulta chiaro che la progettazione e la realizzazione di un sistema esperto richiedono tecniche e metodologie sensibilmente diverse da quelle usate nei sistemi tradizionali. Il tipo di problemi affrontati, generalmente poco strutturabili dal punto di vista formale in partenza e la varietà di tecniche, spesso molto differenti tra loro, offerte dai risultati della ricerca non permettono di affrontare il problema con atteggiamento completamente sistematico.

È chiaro infatti che, quando non sono note a priori le funzionalità desiderate dal sistema nè il tipo di conoscenze che permettono di raggiungerle, non è possibile attivare un progetto di sviluppo software basato su tecniche di ingegneria del software, che, appunto, sono tutte fondamentalmente imperniate sulla disponibilità di un insieme dettagliato di specifiche.
In primo luogo bisogna avviare una analisi il più possibile profonda delle conoscenze mediche coinvolte nei ragionamenti dell'esperto di intossicazioni. Tali conoscenze dovranno essere analizzate sia dal punto di vista medico, per capirne le interconnessioni e la dinamica, sia dal punto di vista informatico, per indagarne la rappresentabilità all'interno di una macchina e le tecniche che più successo hanno avuto a tale scopo.

Dopo questa prima fase di analisi si procederà alla definizione di una ipotesi di formalismo per la rappresentazione della conoscenza e di architettura per il sistema esperto.

3.1 Analisi delle conoscenze

Benchè il termine "conoscenza" sia alquanto poco precisamente definito e ultimamente estremamente abusato, si può affermare che per conoscenza, in questo contesto, si usa intendere un insieme di informazioni organizzate con una certa logica e che vengono spesso utilizzate in modo preciso da chi le possiede. È possibile quindi classificare informalmente i tipi di conoscenze attraverso il criterio organizzativo e le modalità d'uso che le contraddistinguono nella pratica del medico.

É questa una attività in generale molto importante almeno per tre ordini di ragioni:

a) permette di riconoscere tipi di conoscenza già noti nell'ambito delle applicazioni dei sistemi esperti e quindi di rintracciare utili esempi di confronto, oppure di individuare caratteristiche originali di esse. Questo porta ad affrontare nel miglior modo possibile il problema della rappresentazione delle conoscenze, cioè all'individuazione del formalismo e delle tecniche di implementazione più appropriate per renderle utilizzabili da un calcolatore. Si capirà infatti in questa fase se la rappresentazione delle particolari conoscenze coinvolte nel problema specifico richiede una attività di ricerca innovativa nel campo delle tecniche di rappresentazione della conoscenza;

b) permette all'informatico, che in questo contesto viene usualmente chiamato "ingegnere della conoscenza", e all'esperto del campo d'applicazione di stabilire un vocabolario e un linguaggio comuni. Questo aspetto è di particolare importanza per il buon sviluppo del progetto, specialmente per quel che riguarda l'attività sperimentale attorno al prototipo. È chiaro infatti che la velocità di sviluppo dell'attività comune dipende crucialmente dalla facilità di comunicazione tra implementatore del prototipo e sperimentatore. In questa fase l'informatico deve rendersi conto del modo che l'esperto ha di parlare delle proprie competenze e quindi di quale sia la più naturale forma di comunicazione di quelle specifiche competenze. Uno degli obiettivi di ogni progetto di questo genere è fare in modo che, dopo le prime fasi, l'esperto sia in grado autonomamente di far progredire il sistema automatico di supporto, aumentando e aggiornando il bagaglio di conoscenze. È quindi necessario che la forma con la quale le conoscenze vengono comunicate al sistema sia il più vicino possibile a quella usualmente utilizzata direttamente dall'esperto e a lui più gradita;

c) un effetto secondario, ma non poco importante, di questa attività di analisi è il ritorno sulla visione che l'esperto ha delle proprie conoscenze. L'esperto generalmente è abituato a usare le proprie conoscenze piuttosto che a parlarne. Una attività di analisi permette all'esperto stesso di osservare in modo più obiettivo le proprie conoscenze. Questo porta spesso ad una utile revisione e riorganizzazione di alcune delle conoscenze, usualmente le più pratiche e meno discusse del campo.

Sono stati individuati cinque tipi di conoscenza utilizzati concorrentemente dal medico d'urgenza nella sua attività:

. *Conoscenze di interpretazione dei dati*

Lo stato del paziente è noto al medico attraverso i dati provenienti dall'analisi anamnestica, dall'osservazione diretta e dagli esami di laboratorio. Tali dati sono prevalentemente numerici o organizzati in semplici scale di simboli convenzionali. Il ragionamento medico, però, avviene su di un livello più alto; lo stato del paziente infatti viene considerato in termini di concetti e di astrazioni. Le conseguenze di interpretazione dei dati permettono la trasformazione di insiemi di semplici dati in concetti ed astrazioni utili al ragionamento diagnostico e terapeutico.
Il concetto di "insufficienza renale", per esempio, può essere considerato una astrazione che si definisce attraverso l'alterazione correlata di alcuni parametri attraverso i quali si manifesta lo stato del paziente. Il riconoscere una insufficienza renale in un paziente è quindi un'operazione strutturata che consiste nel richiedere un certo insieme di dati di laboratorio, compiere un certo insieme di osservazioni e mettere in relazione in un modo ben preciso i risultati numerici.
Questa attività coinvolge sicuramente una parte importante delle conoscenze del medico, ma può essere considerata in un certo senso più generica, più a basso livello. In altre parole, il ragionamento che porta ad una diagnosi specifica o a delle decisioni terapeutiche si avvale dei risultati dell'applicazione delle conoscenze sull'interpretazione dei dati, ma non è influenzata dai dettagli di tale attività.
In questa classe di conoscenze rientrano anche tutte quelle che permettono al medico di muoversi correttamente all'interno dei protocolli di reparto e di ospedale; le modalità con le quali, per esempio, devono essere richiesti degli esami di laboratorio o vanno prescritti interventi terapeutici routinari.
È interessante notare come, in certe fasi, è necessario condurre contemporaneamente un certo numero di questi "ragionamenti ausiliari", in particolare quando questi implicano verifiche successive a tempi stabiliti.
Dal punto di vista informatico si può dire che queste sono le conoscenze che permettono di passare da sequenze di dati numerici ad asserzioni logiche e a descrizioni ad alto livello.

Conoscenza tassonomica

Si tratta delle conoscenze che permettono di individuare l'agente causa dell'intossicazione (nel nostro caso si tratta di un fungo) in base alle informazioni fornite dal paziente stesso, dai suoi accompagnatori o dal risultato di esami micologici. Le informazioni sulle diverse famiglie di funghi riguardano l'aspetto (colore, odore, forma, etc.), il luogo di ritrovamento (altitudine, umidità, etc.), la costituzione chimica, etc. Tali informazioni sono organizzabili in modo da favorire una veloce classificazione delle notizie, spesso frammentarie e incomplete, che si riescono a cogliere e la formazione di una serie di ipotesi intorno alla specie intossicante. Questa attività di classificazione e riconoscimento deve essere svolta dal medico non appena possibile e rendere disponibili i propri risultati al resto dei ragionamenti.

Conoscenza sulla diagnosi di fase

Come si è detto, attorno alla diagnosi di fase ruota gran parte del ragionamento medico nei casi di intossicazione. Si tratta delle conoscenze necessarie per interpretare le variazioni nello stato del paziente come l'evoluzione di una precisa intossicazione. Queste conoscenze si basano su di una suddivisione convenzionale in periodi delle variazioni delle condizioni del paziente che una intossicazione causa nel tempo. Ognuna delle intossicazioni ha un suo caratteristico andamento a fasi, dove ogni fase può essere riconosciuta in base allo stato di alcuni parametri e delle relazioni tra essi. Le osservazioni che permettono al medico di rendersi conto delle transizioni da una fase all'altra sono condotte applicando conoscenza di interpretazione di dati.
Spesso la diagnosi rimane incerta per molte ore, ed è proprio la verifica di un certo andamento nell'evoluzione dell'intossicazione che permette di riconoscere l'agente tossico. In questi casi il medico deve essere in grado di costruire una interpretazione multipla della evoluzione delle condizioni del paziente: deve, in altre parole, utilizzare concorrentemente la sua conoscenza sulla diagnosi di fase relativa a più forme di intossicazione, fino a che una sola di esse sia congruente ed escluda così le altre.

Conoscenze terapeutiche

Si tratta delle conoscenze che permettono di eseguire correttamente gli interventi terapeutici non appena questi si rendano necessari. Queste competenze suggeriscono al medico osservazioni in base alle quali attivare o meno procedure d'intervento che possono essere localizzate nel tempo o richiedere l'esecuzione periodica di un certo numero di azioni di controllo o di regolazione. Le conoscenze legate al controllo e alla gestione del bilancio idrico del paziente sono un tipico esempio: in base alle condizioni del paziente, quindi alla fase della sua intossicazione, al suo stato generale e agli interventi terapeutici operati fino a quel momento, il medico deve controllare la diuresi forzata attraverso l'applicazione di fleboclisi in quantità opportuna.
Anche la somministrazione periodica di farmaci è controllata attraverso conoscenze di questo tipo.
Un aspetto fondamentale che comunque caratterizza le conoscenze terapeutiche riguarda la duplice natura di tali conoscenze: ogni decisione sugli interventi da effettuare fa riferimento infatti al "cosa" bisogna fare e al "come" bisogna farlo. Questi aspetti sono ambedue fondamentali nell'attività decisionale e, dal punto di vista delle conoscenze, innescano metodologie di ragionamento tra loro molto differenti.

Conoscenza euristica

Costituisce la vera esperienza del medico. Si tratta di quelle conoscenze per lo più apprese attraverso la considerazione di un grande numero di casi prece-

denti. Un buon esempio è costituito dalle conoscenze che permettono di prendere decisioni terapeutiche mettendo in relazione eventuali patologie preesistenti nel paziente e le sue attuali condizioni. Altro tipico esempio sono le conoscenze che permettono al medico di privilegiare l'osservazione di alcuni parametri in casi particolari. Sono certamente conoscenze frammentarie e indipendenti che devono intervenire nei processi decisionali in modo del tutto libero e generale.

4. PROBLEMI DI RAPPRESENTAZIONE DELLA CONOSCENZA

Dall'analisi delle conoscenze mediche coinvolte nel trattamento dell'intossicazione da fughi a lunga incubazione appare chiaro che ogni tentativo di rappresentazione deve misurarsi con un difficile problema di fondo causato dalla necessità di sfruttare concorrentemente conoscenze di tipo disomogeneo.
Come si è visto, il medico utilizza conoscenze classificabili sostanzialmente in cinque categorie: di interpretazione dei dati, tassonomiche, sulla diagnosi di fase, terapeutiche ed euristiche. Tali conoscenze intervengono tutte concorrentemente nell'attività decisionale che viene condotta dal medico durante il trattamento dei singoli casi di intossicazione. Nonostante la complessità di interazione tra tipi di conoscenza tra loro così differenti per natura, possiamo applicare una distinzione più generale su questi tipi di conoscenze in modo da ottenere due grandi classi: conoscenze di carattere procedurale e conoscenze di carattere dichiarativo.
Si tratta di una distinzione molto nota negli ambienti di ricerca dell'Intelligenza Artificiale e ben indagata da parte di studiosi noti [2].

Uno dei motivi che hanno portato al successo i sistemi esperti sta proprio nel fatto che la rappresentazione a regole (decisamente dichiarativa) delle conoscenze dell'esperto ha permesso di esprimere e strutturare conoscenze tra loro slegate o frammentarie o incomplete ed euristiche in modo unitario nella base della conoscenza. I sistemi esperti basati su regole sono stati applicati con successo proprio in quei domini caratterizzati da conoscenze di questo tipo.

Esistono comunque domini di competenze in cui in modo esplicito vengono utilizzate conoscenze decisamente procedurali dagli esperti; questo tipo di conoscenze riguarda sequenze di azioni o di fatti che devono essere noti o eseguiti per risolvere un problema. La rappresentazione dichiarativa di queste conoscenze che per loro natura sono legate in modo procedurale non solo è spesso difficoltosa dal punto di vista della formalizzazione, ma rende più arduo sia il processo di aquisizione che quello di incremento delle conoscenze. Inoltre, spesso l'esperto trova poco comprensibile questa trasformazione.

Nonostante l'atteggiamento dichiarativo di rappresentazione sia sempre stato "vincente" nella realizzazione di sistemi esperti, alcuni studi affiorati negli anni più recenti e rintracciabili nella letteratura sull'Intelligenza Artificiale [3] [4] hanno rivelato la possibilità di realizzare sistemi esperti che impiegano rappresentazioni della conoscenza sia procedurali che dichiarative. Per quel che concerne l'uso di formalismi di rappresentazione di tipo dichiarativo, l'esperienza della ricerca in Intelligenza Artificiale in questa direzione offre soluzioni non circoscrivibili alle sole regole: anche in questo caso l'uso di formalismi più vicini alla natura delle conoscenze coinvolte è già stato approcciato [6] e nel nostro lavoro è stata presa la stessa direzione.

Il problema della rappresentazione della conoscenza impiegata nel trattamento delle intossicazioni da funghi velenosi a lunga incubazione diviene quindi quello di tradurre in formalismi ad hoc tutti i tipi di conoscenze analizzate e più sopra descritti.

4.1 Scelte di rappresentazione della conoscenza

Per quel che riguarda la conoscenza procedurale, possiamo assimilare ad essa la conoscenza relativa alla diagnosi di fase. Infatti, anche se l'aspetto rilevante è costituito dalla verifica di condizioni e non dall'esecuzione di azioni, non c'è dubbio che tali verifiche vadano condotte sequenzialmente e con un insieme limitato di alternative per ogni passo.

Dato che in gran parte dei ragionamenti del medico sono fortemente legati ad una serie di considerazioni che riguardano esplicitamente la fase, è necessario che un sistema automtico di supporto disponga di una rappresentazione esplicita degli stadi di evoluzione a fasi della malattia: in altre parole, è necessario poter conoscere in ogni istante il punto di vista delle conoscenze sulla evoluzione a fasi dell'intossicazione.
Lo strumento di rappresentazione per la conoscenza procedurale che dovremo scegliere dovrà avere, tra l'altro, anche la caratteristica di rendere disponibili in modo dinamico, informazioni relative allo stato di avanzamento delle procedure rappresentate.

Il formalismo che ci è sembrato più adatto allo scopo di rappresentare questo tipo di conoscenza è quello dei "GRAFI DI EVENTI". Si tratta di un modo di rappresentare conoscenze procedurali derivato dalle reti di Petri [9] e messo a punto, in una prima versione, nell'ambito di un progetto di ricerca volto alla realizzazione di un sistema per il controllo e la diagnostica nei processi industriali [4].
In particolare, il progetto riguardava un sistema in grado di verificare in modo continuo il buon funzionamento del ciclo acqua-vapore in una centrale termoelettrica. Anche in quel caso, la principale attività "intelligente" del sistema era quella di confrontare, in caso di malfunzionamento, l'evoluzione dello stato dell'impianto con una serie di comportamenti, tipica-

mente causati da guasti, suggeriti dagli esperti operatori di centrale.

I Grafi di Eventi sono derivati dal formalismo delle reti di Petri, come strumento di rappresentazione di processi concorrenti. Le reti di Petri si sono rivelate di grande utilità nello studio in generale delle problematiche delle attività che si svolgono in parallelo ma che interagiscono tra di loro in modo più o meno sostanziale. Le applicazioni di questi studi sono numerose in campo informatico tecnico: programmazione e progettazione di elaboratori paralleli, metodologie di sviluppo software, e, anche più in generale, organizzazione aziendale, pianificazione dei processi di produzione industriale e addirittura modellazione di processi tipici della fisica moderna.

Dalle reti di Petri, i Grafi di Eventi ereditano, più che altro, la rappresentazione grafica ed alcune delle regole di comportamento. Per i Grafi di Eventi non esiste al momento invece una trattazione formale rigorosa. Per gli scopi di questo progetto, tale trattazione non sembra essere necessaria e quindi non verranno affrontati i complessi problemi teorico-matematici che inevitabilmente si presenterebbero. Tuttavia, è doveroso sottolineare come una precisa formulazione teorica di questo formalismo potrebbe portare interessanti ripercussioni nella ricerca nel campo della rappresentazione della conoscenza in generale.

Un Grafo di Eventi è un grafo orientato bipartito; i tipi di nodi sono due: i posti, che rappresentano le componenti dello stato del sistema e che graficamente vengono rappresentati con cerchi e gli eventi, che descrivono appunto gli eventi che costituiscono l'evoluzione del sistema e che graficamente vengono rappresentati con rettangoli. Gli archi orientati possono connettere posti con eventi o eventi con posti, ma non posti con posti o eventi con eventi (fig. 1). Associate ad ogni evento sono due espressioni: una di queste rappresenta la condizione dell'ambiente che determina l'evento stesso e l'altra l'azione che l'evento causa sull'ambiente.
Ogni posto può essere marcato, ed in questo caso il posto è rappresentato con un punto nero inscritto. In ogni istante possono essere marcati un numero arbitrario di posti. L'insieme dei posti marcati si dice "marcatura del grafo". La dinamica del sistema viene descritta mediante l'evoluzione della marcatura.

Ad ogni grafo di eventi è associata una marcatura iniziale che determina lo stato dal quale ha inizio l'evoluzione del grafo stesso. La marcatura viene fatta evolvere dallo scatto degli eventi. Un evento può scattare quando: tutti i posti per cui esiste un arco da essi all'evento (posti di input) sono marcati, tutti i posti per cui esiste un arco dell'evento ad essi (posti di output) non sono marcati e la condizione associata all'evento è verificata dall'ambiente.
Lo scatto di un evento toglie la marcatura da tutti i suoi posti di input e marca tutti i suoi posti di output.

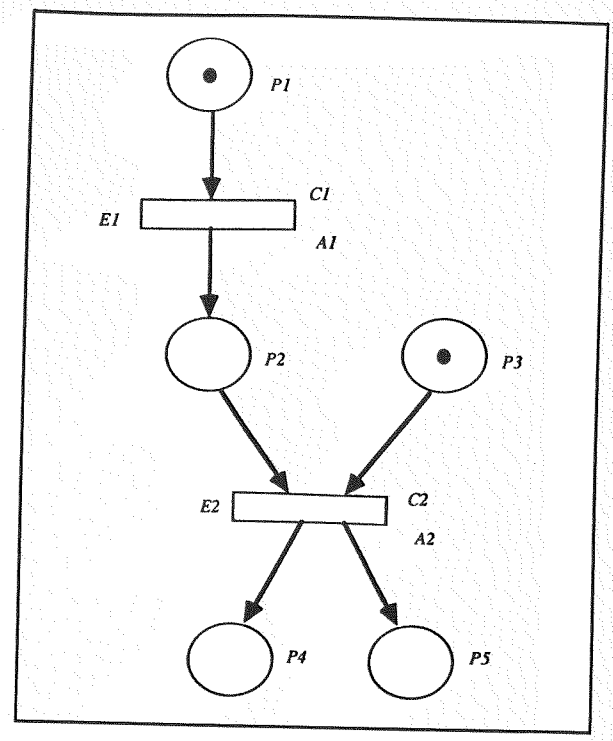


Fig. 1

Nell'esempio della fig. 1 la marcatura iniziale è nei posti P1 e P3. L'evento E1 ha il suo unico posto di input (P1) marcato e il suo posto di output (P2) non marcato, quindi può scattare non appena la condizione C1 è verificata. La mancanza di marca in P2 impedisce invece all'evento E2 di scattare.
Lo scatto dell'evento E1 toglie la marca in P1 e la pone in P2, nello stesso tempo viene eseguita l'azione A1. A questo punto, se la condizione C2 è verificata, l'evento E2 può scattare togliendo le marche da P2 e P3 e marcando P4 e P5; in questo modo l'azione A2 viene eseguita.
La marcatura corrente descrive quindi in modo più completo lo stato di evoluzione del Grafo di Eventi e quindi delle conoscenze rappresentate e allo stesso tempo determina completamente l'evoluzione futura del grafo. Questa rappresentazione esplicita e accessibile dello stato del grafo permette al resto del sistema, di osservare dinamicamente il comportamento del grafo stesso o di influenzare il comportamento osservando o modificando direttamente la marcatura.
In fig. 2 uno schematico esempio di un Grafo di Eventi che rappresenta l'evoluzione a fasi dell'intossicazione da Amanita Phalloides, dove gli eventi sono le varie fasi dell'intossicazione stessa.

Tra i cinque tipi di conoscenze individuati e più sopra descritti possiamo constatare che le conoscenze tassonomiche hanno carattere esplicitamente dichiarativo. Dal punto di vista della rappresentazione, il problema è quello di esplicitare il più possibile la struttura tassonomica stessa di questo tipo di conoscenza.
Quello della conoscenza strutturata è un problema

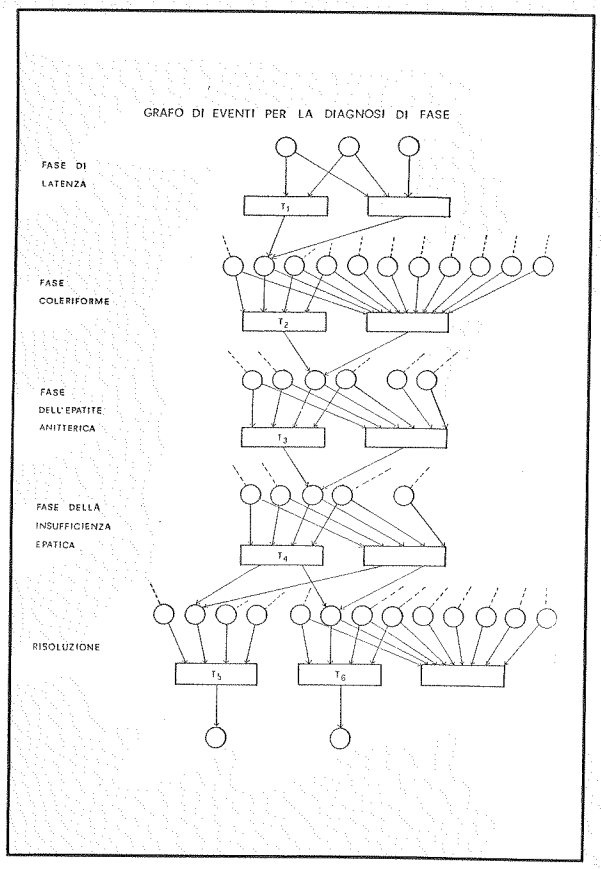


Fig. 2

ben noto nell'ambito delle applicazioni dell'Intelligenza Artificiale e si può affermare che ogni sistema di una certa rilevanza prevede l'uso di un formalismo di questo tipo [10].
Nel nostro caso la struttura delle informazioni da rappresentare sembra essere piuttosto semplice e adatta ad una rappresentazione frame-like [11].
Il problema del riconoscimento dell'agente tossico, che è fondamentale nella prassi del medico del pronto soccorso, abbiamo già visto essere riconducibile ad una tipica struttura di confronto e riempimento di slots. In fig. 3 l'esempio di una struttura tassomica di questo tipo per l'identificazione dell'Amanita Phalloides.

Per quel che riguarda le conoscenze per l'interpretazione dei dati e la loro rappresentazione, in parte si può parlare decisamente di procedure: come si è detto, tali conoscenze devono essere applicate in modo ben organizzato rispetti ai ragionamenti più ad alto livello in corso (per esempio, la diagnosi di fase) e quindi devono essere rappresentate in modo il più possibile omogeneo rispetto ad essi.

In alcuni casi le conoscenze di interpretazione di dati sono invece frammentarie e non strutturate. Spesso infatti la correlazione fra dati diversi può essere naturalmente sotto forma di regole sparse. Una regola, in questo caso, è una coppia condizione-azione. La condizione esprime la relazione tra i parametri e l'a-

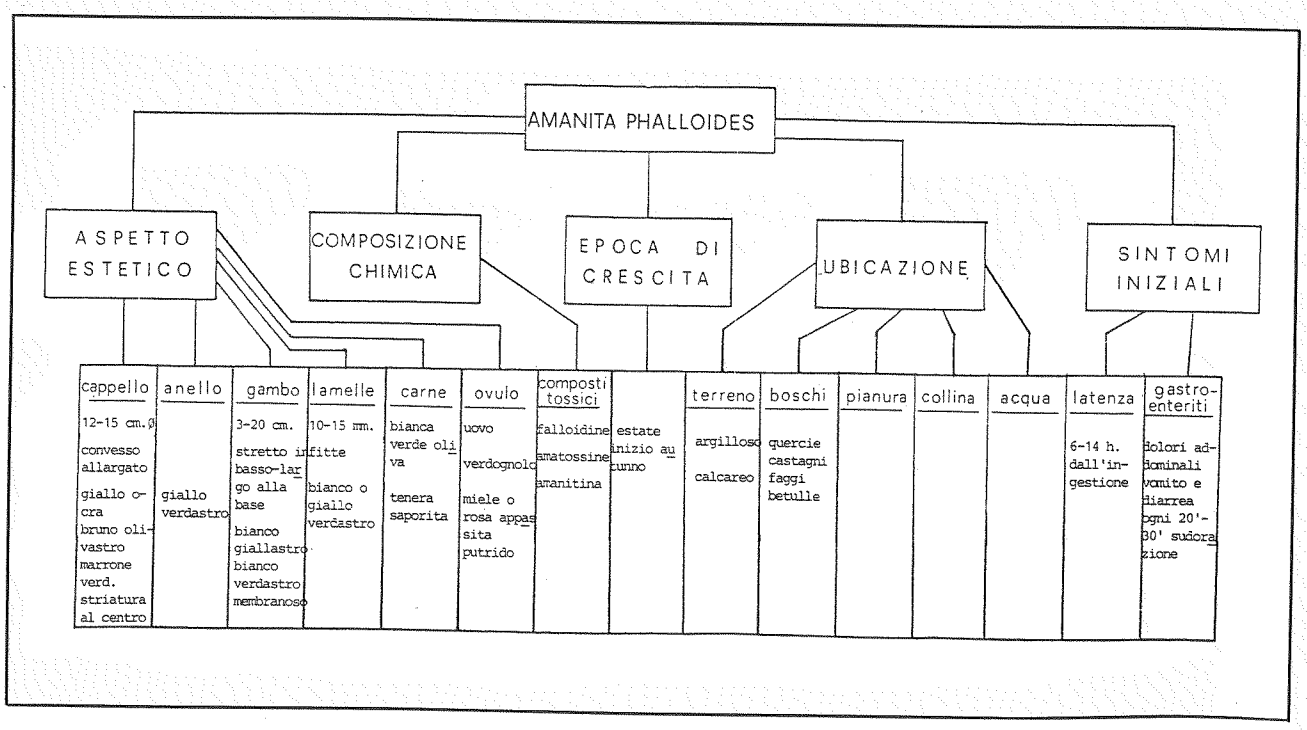


Fig. 3

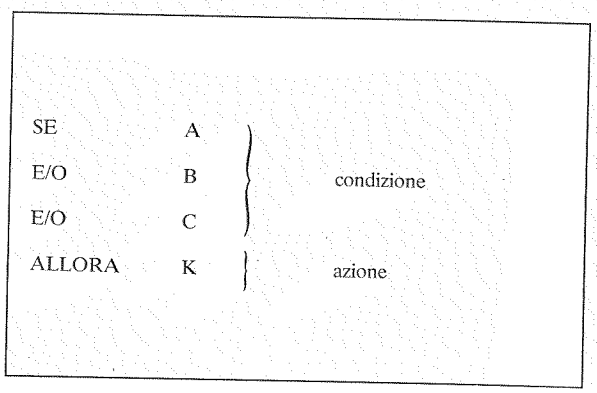


Fig. 4

zione asserisce il concetto che riassume tale relazione (fig. 4). Le regole di questo tipo devono essere applicate in modo non deterministico, ovvero ogni regola deve poter eseguire la propria azione non appena sia verificata la propria condizione. Non esiste quindi ordinamento o sequenzializzazione nell'applicazione di questo tipo di regole.

Per le conoscenze terapeutiche che permettono di eseguire correttamente gli interventi non appena questi si rivelano necessari, vale il discorso fatto per quelle relative alla interpretazione dei dati. In parte si può parlare di conoscenza di tipo procedurale con particolari esigenze di rappresentazione (fig. 5), in parte di conoscenza rappresentabile sotto forma di coppie condizione-azione con attivazione non deterministica (fig. 6). È interessante notare che, in questo caso, le condizioni farebbero comunque riferimento a risultati ottenuti applicando le conoscenze relative alla interpretazione dei dati, alla diagnosi di fase e alla classificazione dell'agente tossico, piuttosto che ai dati provenienti dal paziente.

Infine, le conoscenze euristiche hanno caratteristiche di indipendenza e frammentarietà e devono intervenire nei processi decisionali in modo del tutto libero e generale (fig. 7). Il modo più naturale per rappresentarle è sicuramente attraverso le regole condizione-azione. Anche in questo caso si tratta di conoscenza che agisce in base allo stato di applicazione di altre conoscenze. In questo caso le azioni influenzano proprio l'applicazione di altre conoscenze suggerendo semplificazioni o parametrizzazioni.

5. CONCLUSIONE

L'esigenza di rappresentare con formalismi adeguati conoscenze tra loro differenti ha portato all'analisi delle conoscenze coinvolte in uno specifico ambito. I principali tipi di conoscenze ricavate dall'analisi ha condotto alla definizione di due classi di tipi di conoscenze utilizzate dall'esperto nella diagnosi e nel trat-

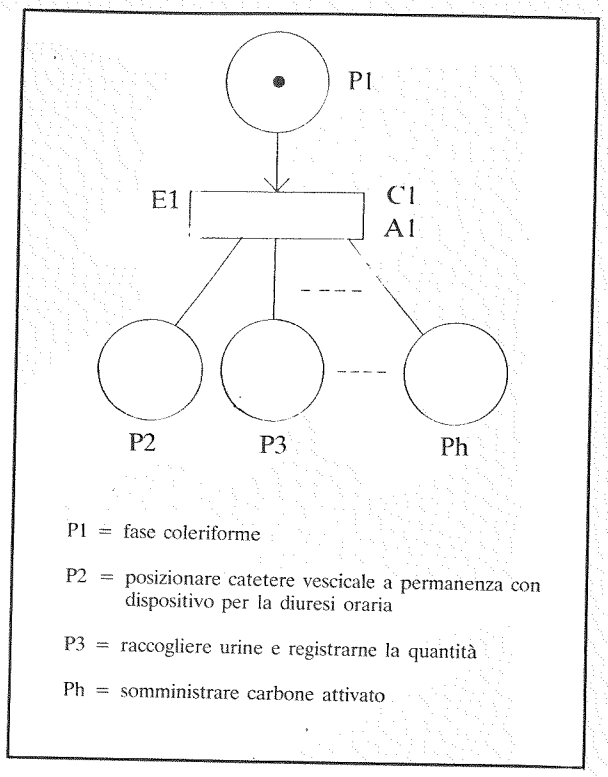


Fig. 5

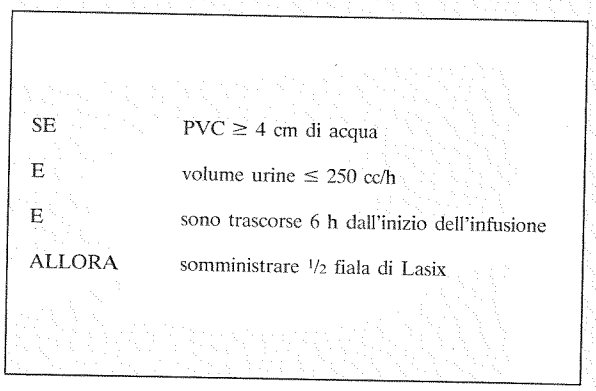


Fig. 6

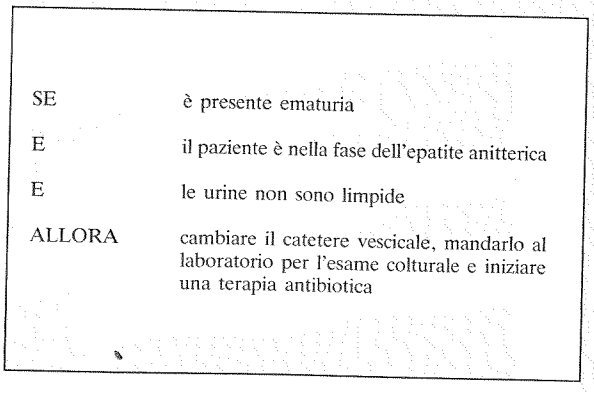


Fig. 7

tamento di un insieme di sindromi: conoscenze di tipo procedurale e conoscenze di tipo dichiarativo. Sono poi stati introdotti formalismi di rappresentazione sia dichiarativi che procedurali.

6. RINGRAZIAMENTI

Vogliamo qui ringraziare il Prof. D. Costantino per la paziente e assidua collaborazione nel processo di acquisizione e analisi delle conoscenze, la D.ssa S. Cartassegna e la D.ssa O. Pancolini per il supporto alla comprensione del problema, S. Baglini e L. Galletti per l'impegno e l'entusiasmo mostrati nella parte implementativa del prototipo. Un ringraziamento anche al Prof. G. Degli Antoni per gli spunti indispensabili che ci ha fornito e per la supervisione del lavoro.

7. RIFERIMENTI BIBILIOGRAFICI

[1] F. Hayes-Roth, D.A. Waterman, D.B. Lenat, BUILDING EXPERT SYSTEMS, Addison Wesley Publishing C., Readings, Mass., 1983

[2] T. Winograd, FRAME REPRESENTATIONS AND DECLARATIVE/PROCEDURAL CONTROVERSY, in "Representation and Understanding: Studies in Cognitive Science", D.G. Bobrow and A.M. Collins (eds.), Academic Press, New York, 1975

[3] U. Bonollo, M. Georgeff, PROCEDURAL EXPERT SYSTEMS, Proc. IJCAI n. 8, Karlsruhe, 1983

[4] M. Gallanti, G. Guida, L. Spampinato, A. Stefanini, REPRESENTING PROCEDURAL KNOWLEDGE IN EXPERT SYSTEMS: AN APPLICATION TO PROCESS CONTROL, Proc. IJCAI n. 9, Los Angeles, 1985

[5] S. Amarel, EXPERT BEHAVIOUR AND PROBLEM REPRESENTATION, in "Artificial and Human Intelligence", A. Alithorn, R. Benerji (eds), North Holland, 1984

[6] J.S. Aikins, A REPRESENTATION SCHEME USING BOTH FRAMES AND RULES, in "Rule-Based Expert Systems", B.G. Buchnan, E.H. Shortliffe (eds.), Addison Wesley, Readings, Mass., 1984

[7] D. Costantino, G. Fioni, J.W. Shroder, DIAGNOSI DELLA INTOSSICAZIONE DA AMANITA PHALLOIDES, in "La Diagnosi e la Terapia degli Avvelenamenti da Funghi, Atti del Corso di Aggiornamento per gli Operatori Sanitari di pronto Soccorso della Regione Lombardia, Milano, 1982

[8] D. Costantino, GLI AVVELENAMENTI DA FUNGHI A LUNGA INCUBAZINOE, Edizione a cura della FARMITALIA, Milano, 1985

[9] W. Reisig, PETRI NETS: AN INTRODUCTION, Springer Verlag, Berlino, 1985

[10] N.V. Findler (ed), ASSOCIATIVE NETWORK, Academic Press, New York, 1979

[11] M. Minsky, A FRAMEWORK FOR REPRESENTING KNOWLEDGE, in J. Hangeland (ed) "Mind Design", MIT Press, Cambridge, MA, 1981

Questo lavoro è stato presentato al 6° Corso Convegno Internazionale Radiodiagnostica e Terapie Integrate in Oncologia (Roma 17-22 feb. 1986) sul Suppl. n°1 del vol. 11/1 di RAYS, ed è qui pubblicato per gentile concessione del Direttore del Corso prof. A. Romanini.
La ricerca in corso è stata patrocinata da CNR nell'ambito del Progetto Strategico Sistemi Esperti in Medicina.

UN SISTEMA ESPERTO INTEGRATO CON MODULI DI GESTIONE E RAPPRESENTAZIONE DATI PER IL MONITORAGGIO POST-OPERATORIO DI SOGGETTI SOTTOPOSTI A BY-PASS DIGIUNO-ILEALE.

Elena Lattuada, Paola Macchi
Dipartimento di Scienze dell'Informazione - Università di Milano

Santo B. Doldi, Enzo Lattuada
Istituto di Clinica Chirurgica III e Chirurgia Plastica - Università di Milano

RIASSUNTO

In questo lavoro si illustra un prototipo di un sistema per il monitoraggio post-operatorio di pazienti obesi sottoposti a by-pass intestinale.
Tale strumento, che è in grado di fornire un quadro evolutivo complessivo di ogni soggetto, fa uso di una shell di sistema esperto, Micro Expert, come modulo di un sistema più ampio finalizzato anche all'archiviazione dei dati clinici rilevanti ed alla rappresentazione grafica dell'evoluzione nel tempo di ogni paziente.

ABSTRACT

This paper describes a project and its implementation of a system for monitoring obese patients after intestinal by-pass. The architecture of this system is structured by a set of modules and one of this is an expert system: it's built by mean of a shell named Micro Expert.

1. INTRODUZIONE

Il progetto in fase di realizzazione presso l'Istituto di Cibernetica in collaborazione con la III Clinica Chirurgica dell'Università degli Studi di Milano riguarda un sistema per il monitoraggio post-operatorio di pazienti obesi sottoposti a by-pass digiuno-ileale.

L'intervento consiste nell'esclusione del 90% circa dell'intestino tenue da transito alimentare e il malassorbimento intestinale indotto è la causa principale del notevole calo ponderale che si riscontra nei mesi successivi all'intervento.

Un sistema rivolto a questo ambito è uno strumento che, basandosi sui tests di laboratorio effettuati periodicamente dai pazienti nei 2/3 anni che seguono l'operazione chirurgica, sia in grado di fornire un quadro complessivo dell'evoluzione clinica di ogni soggetto.
Tale strumento che fa uso di una shell di sistema esperto, Micro Expert, è finalizzato anche all'archiviazione dei dati clinici e alla rappresentazione grafica dell'andamento nel tempo di ogni paziente.

2. IL PROBLEMA MEDICO

L'obesità grave essenziale è una condizione patologica caratterizzata da un eccessivo aumento dell'adipe e da una conseguente notevole eccedenza nel peso corporeo. Si tratta di una malattia che con l'invecchiamento del paziente viene a determinare importanti alterazioni di organi ed apparati vitali, compromettendone la normale funzionalità e causando l'insorgenza di stati patologici gravi difficilmente controllabili.
Nei casi in cui la terapia medica e dietetica non porti a risultati validi e stabili diventa consigliabile ricorrere ad un trattamento chirurgico quale il by-pass intestinale.

Requisiti fondamentali per l'intervento sono:

- Peso superiore al peso ideale di almeno il 60% negli uomini e di almeno l'80% nelle donne.
- Età non superiore a 55 anni e non inferiore ai 18 anni.
- Obesità essenziale, esogena, non discrinica.
- Completa disponibilità psicologica del paziente.
- Incapacità a vivere ed accettare il proprio corpo ed assoluta necessità di svolgere un'attività lavorativa normale.

Il by-pass intestinale digiuno-ileale determina un malassorbimento intestinale globale e una conseguente diminuzione dell'assorbimento dei grassi e degli zuccheri e quindi, in ultima analisi, un sensibile calo ponderale.
La tecnica chirurgica è tale da conservare in situ defunzionalizzato la maggior parte dell'intestino tenue permettendo perciò, se necessario, il ripristino del normale transito intestinale.

Se da un lato il malassorbimento intestinale indotto, che causa la sindrome dell'intestino corto da eslcusione intestinale estesa, è tale da permettere il raggiungimento di uno stabile decremento ponderale, dall'altro parenchimi epatico, renale e apparato scheletrico possono subire alterazioni rilevanti, sia nella fase iniziale che a distanza di tempo dall'intervento chirurgico.

Al fine di prevenire o comunque trattare tempestivamente le complicazioni che possono conseguire ad un by-pass intestinale è assolutamente indispensabile l'attuazione di un attento e costante controllo clinico ambulatoriale del paziente operato.

La frequenza di tali visite è solitamente mensile durante la fase di massimo dimagramento, che avviene nei primi 12 mesi dopo l'intervento chirurgico, bimestrale fino al secondo anno e semestrale dal terzo anno in poi.
Ogni paziente si presenta al controllo dopo aver effettuato una serie di esami ematochimici riguardanti tra l'altro i principali parametri di funzionalità epatica e renale, il quadro elettrolitico plasmatico, il quadro lipemico e proteico e la crasi ematica.

Il follow-up ambulatoriale consiste non solo nella valutazione degli esami ematochimici eseguiti dal paziente, ma anche in un'accurata visita che permette di definire il quadro clinico complessivo dell'obeso tale da confermare o modificare in modo opportuno la terapia sostitutiva in corso (elettrolitica, minerale, vitaminica ed epatoprotettrice) e l'eventuale prescrizione di esami strumentali particolari o consulenze specialistiche necessarie.
Seguendo questo indirizzo di lavoro si vengono a ridurre in modo sensibile l'incidenza e l'entità di complicanze cliniche riconducibili al by-pass digiuno-ileale, [4], [5],.

3. SISTEMI ESPERTI E MICRO EXPERT

I sistemi esperti sono strumenti in grado di riprodurre i processi logico-deduttivi che l'esperto umano, utilizzando le sue conoscenze e la sua esperienza, attua nella soluzione di un particolare problema.
La loro caratteristica principale è perciò la capacità di trarre conclusioni da un insieme di informazioni relative ad una circoscritta area conoscitiva codificate in una struttura detta base della conoscenza.
In tale struttura trova quindi posto, sotto forma di regole, quello che è il patrimonio conoscitivo di un esperto che nella maggioranza dei casi è costituito di due componenti, quella nozionistica e quella euristica, [3].

Strutturare e codificare questo tipo di regole è uno stadio fondamentale nella costruzione di un sistema esperto, [2].
Altra componente essenziale di un sistema esperto è il motore inferenziale basato su un algoritmo di ricerca e selezione che interagendo sia con la base della conoscenza che con i dati correnti del problema, seleziona un certo numero di regole utili alla soluzione del caso in esame, [1].
Un sistema esperto "svuotato", cioè privato dell'insieme di regole che costituiscono la base della conoscenza, diventa una "shell".

Strumenti di questo tipo, oggi disponibili anche per personal computer, costituiscono un valido punto di partenza per lo sviluppo di sistemi esperti: si tratta di strutture polivalenti utilizzabili in diversificati settori di conoscenza che pongono come unico vincolo la compatibilità tra la struttura logica del problema in esame e quella della struttura di derivazione del sistema.
Un esempio è Micro Expert, una shell che simula il processo di ragionamento di un esperto facendo riferimento ad un modello che gli viene fornito in un particolare linguaggio, l'Advice Language.

Il modello o base della conoscenza, consiste di un certo numero di regole collegate tra loro in modo da formare uno o più alberi alla radice dei quali stanno i goal, cioè le ipotesi di cui il sistema deve stabilire il grado di certezza, e alle cui foglie stanno le domande da porre all'utente.
Come il sistema esperto da cui è derivato (Prospector), anche Micro Expert si compone di un motore inferenziale che fa uso sia del meccanismo di backward che di forward chaining; il sistema discende l'albero corrispondente al goal selezionato fino a raggiungere le foglie, tramite le quali assume informazioni dall'utente.
Ottenute le risposte alle domande selezionate risale all'indietro le catene inferenziali che costituiscono l'albero stesso attribuendo così alla radice un fattore

di certezza che rappresenta la veridicità del goal per i dati correnti.
Nell'albero i nodi intermedi, oltre a rappresentare i passi del processo inferenziale attraverso i quali si giunge alla valutazione del goal, permettono di definire il tipo di manipolazione eseguito sul valore dell'antecedente nella catena logica.

Il sistema tratta ogni ipotesi, regola o goal in termini probabilistici ed associa quindi ad ogni nodo dell'albero un valore numerico calcolato utilizzando quello dell'antecedente in accordo o con la logica matematica classica o con la teoria bayesiana o con speciali funzioni che mappano numeri reali di probabilità.
I fattori di certezza sopra citati rappresentano la misura di veridicità associata ad ogni goal o ipotesi, il cui range di variabilità è compreso tra −5 e +5 dove −5 significa "assolutamente falso", +5 "assolutamente vero", 0 esprime incertezza totale ed i valori intermedi costituiscono tutte le altre possibili valutazioni.

4. ARCHITETTURA DEL SISTEMA

Il sistema in fase di realizzazione è composto da un nucleo centrale costituito da Micro Expert, opportunamente modificato in base alle esigenze del particolare problema medico, e da due moduli specifici finalizzati a migliorare l'interfaccia con l'utente ed a consentire l'archiviazione e il reperimento in tempi successivi dei dati significativi.

Nodo essenziale nello sviluppo dell'architettura del sistema è stata l'esigenza imposta dall'ambito del problema di lavorare in ambiente omogeneo nel quale poter immagazzinare e reperire informazioni cliniche, richiedere la valutazione da parte del sistema dei dati già archiviati e poter visualizzare graficamente l'andamento del soggetto in riferimento ad ogni patologia considerata.

La Fig. 1 mette in luce l'architettura del sistema che ha come nodo primario una struttura di memorizzazione e reperimento di dati che oltre a svolgere queste funzioni più tipiche funge da veicolo di comunicazione tra l'archivio, l'expert system ed il modulo grafico.

Infatti l'ambiente medico in cui il sistema si inserisce ha evidenziato subito la necessità, di creare una cartella clinica computerizzata specifica in cui memorizzare i dati significativi per il follow-up post-operatorio dei pazienti sottoposti a by-pass. Dato l'alto numero di casi finora trattati con tale tecnica chirurgica, la durata piuttosto elevata del periodo di monitoraggio e il numero di esami con cui ogni paziente si presenta alle visite ambulatoriali, la mole di dati a disposizione è diventata tale da non consentire una semplice gestione manuale; strettamente collegata a tale esigenza di archiviazione vi è poi quella di reperimento, in tempi successivi, dei dati già memorizzati.

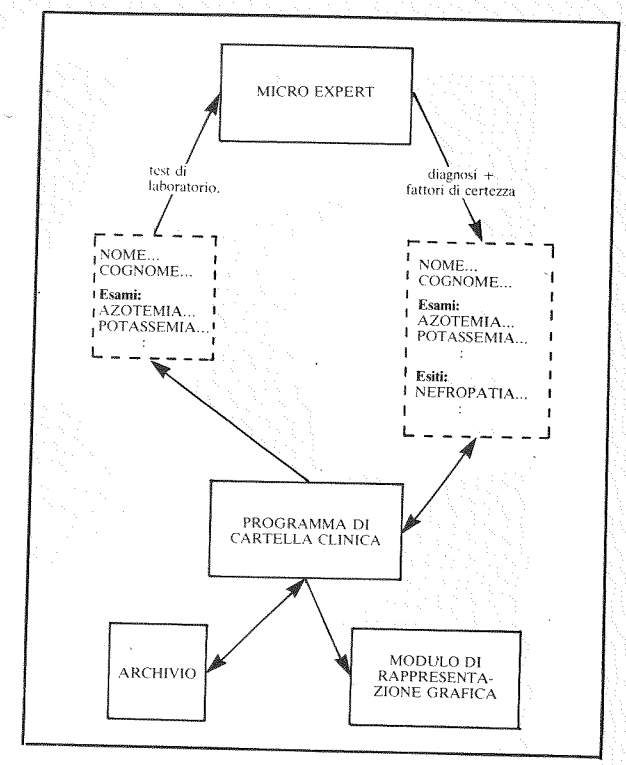


Fig. 1 Architettura del sistema

Entrambe le possibilità, che corrispondono rispettivamente alla selezione delle opzioni 1 e 4 nel menù principale del sistema (Fig. 2), sono comunque mirate a consentire l'integrazione del modulo intelligente (Micro Expert).
I dati considerati rilevanti a questo scopo, e quindi richiesti dal modulo di cartella clinica, sono sia di tipo anagrafico che di tipo clinico. I primi riguardano un certo numero di informazioni generali sul paziente (Fig. 3), mentre i secondi riguardano esclusivamente gli esami clinici obiettivi (test di laboratorio sui principali parametri ematochimici) con cui i pazienti si presentano ai controlli ambulatoriali dopo l'intervento chirurgico (Fig. 4). Altro aspetto è che i dati anagrafici considerati vengono richiesti solo quando il si-

1) INSERIMENTO DATI
2) CONSULTAZIONE EXPERT SYSTEM
3) RAPPRESENTAZIONE GRAFICA
4) REPERIMENTO DATI
5) FINE SESSIONE

Scegliere l'opzione indicandone il numero:

Fig. 2 Menù principale del sistema

COGNOME:
NOME:
SESSO (f/m):
ETÀ:
LUOGO DI NASCITA:
DOMICILIO (via e num.):
CITTÀ:
PROVINCIA (sigla):
TEL.:
DATA INTERVENTO:
TIPO DI INTERVENTO:
PESO:
LUNGH. ANSE DERIVATE:

Visita numero:

Fig. 3 Richiesta dati anagrafici

GLICEMIA:
PROTOMBINEMIA:
BILIRUBINA D.:
SIDEREMIA:
TRIGLICERIDI:
EMATOCRITO:
VOL. GLOBULARE:
PROTIDEMIA:
POTASSIO:
PH URINE:
CREATININEMIA:
PROTEINURIA:
CILINDRI:
CRISTALLINURIA:
RATIO A/G:
BILIRUBINA:
BILIRUBINA I.:
COLESTEROLO:
ERITROCITI:
EMOGLOBINA:
ALBUMINA:
CALCIO:
MAGNESIO:
AZOTEMIA:
URINOCULTURA:
EMOGLOBINURIA:
GLICOSURIA:

Visita numero:

Fig. 4 Richiesta esami di laboratorio

stema stesso accerta, a partire dal cognome del soggetto in esame, che si tratta della prima visita post-operatoria.
Il modulo di reperimento dati è di conseguenza finalizzato a prospettare, a fronte del cognome del paziente, i dati anagrafici relativi e, per ogni visita di controllo effettuata, i risultati degli esami di laboratorio e l'esito della consultazione del sistema esperto.

L'utilizzo di Micro Expert in tale contesto ha suggerito di associare ad ogni goal un possibile stato patologico che può insorgere nei soggetti sottoposti a by-pass intestinale nel periodo post-operatorio e di assumere come nodi foglia le richieste degli esami ematochimici utili per la determinazione delle diagnosi.

La consultazione di Micro Expert, è intrinsecamente interattiva, cioè basata su un ciclo del tipo: selezione del goal da indagare, risposta diretta da parte dell'utente alle domande poste dal sistema, visualizzazione da parte di quest'ultimo del fattore di certezza associato al goal in esame, ritorno al livello di selezione per consentire di scegliere un nuovo goal da indagare o di uscire dalla sessione. Essa è risultata però poco adatta al contesto in esame e alle esigenze di integrazione di cui si è parlato.

Si è resa perciò necessaria un'operazione di alterazione delle prestazioni di Micro Expert, resa possibile dalla definizione e dall'utilizzo di moduli specifici scritti in un linguaggio ad alto livello, quale il ProPascal, con i quali il sistema è interfacciabile.

Il loro uso è finalizzato essenzialmente a due scopi:

- eliminare in fase di consultazione l'interattività di Micro Expert, facendo in modo che esso reperisca i dati necessari direttamente dalla cartella clinica relativa al paziente in esame e che attribuisca automaticamente un fattore di certezza a tutti i goal presenti,

- archiviare nella cartella clinica i risultati della consultazione del sistema esperto in modo da avere un quadro completo dell'andamento post-operatorio di ogni paziente e da consentirne la rappresentazione grafica rispetto al tempo.

L'archivio risulta quindi, dopo la sessione del sistema esperto, aggiornato con le probabilità di presenza di una diagnosi; è dall'interazione con questi ultimi dati che il modulo di rappresentazione grafica riesce a fornire per ogni paziente un quadro sintetico ed immediato dell'andamento dei fattori di certezza relativi a ciascuna patologia. Un esempio è riportato in Fig. 5.

La cartella clinica citata viene periodicamente aggiornata con i dati anagrafici e gli esiti dei test di laboratorio di ogni soggetto; così come il medico trae da questi dati indicazioni significative sull'insorgere di

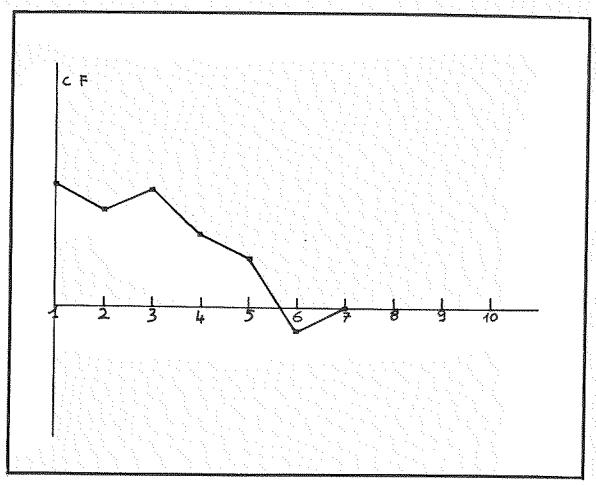


Fig. 5 Grafico dei fattori di certezza nel tempo

eventuali complicanze, Micro Expert, richiamato con l'opzione 2 di Fig. 2, trasforma il risultato di ogni esame, di un dato paziente per una specifica visita, in probabilità di anomalia rispetto ad un andamento ideale del valore stesso ed attribuisce, a partire dall'insieme dei dati di laboratorio, un fattore di certezza ad ogni goal presente nel modello.

La visualizzazione grafica dei fattori di certezza di ogni goal costituisce quindi una sintetica rappresentazione del quadro clinico, nella sua evoluzione postoperatoria, ed un momento riassuntivo delle prestazioni del sistema.

5. CONCLUSIONI

Terminata la realizzazione, i momenti successivi del lavoro prevedono una fase di analisi di sensibilità e di validazione, da parte dello staff medico, delle prestazioni del sistema con particolare riferimento al modulo "intelligente".

6. RINGRAZIAMENTI

Si ringraziano il Prof. G. Degli Antoni e il Prof. W. Montorsi per la collaborazione prestata, e la Dott. S. Bandini per i validi suggerimenti forniti in fase di progettazione.

7. BIBLIOGRAFIA

[1] B.G. Buchanan, E.H. Shortliffe, RULE BASED EXPERT SYSTEM, Addison-Wesley Publishing Company, 1984.

[2] F. Hayes-Roth, D.A. Waterman, D.B. Lenat, BUILDING EXPERT SYSTEM, Addison-Wesley Co., MA, 1983

[3] S. Bandini, I SISTEMI ESPERTI, Quaderni di Informatica, Vo. LXXII, N. 7, 1985

[4] V. Scaravaggi, P.M. Vita, E. Lattuada, A. Apollo, G. Spezzano, ASPETTI DEL CONTROLLO AMBULATORIALE DEI PAZIENTI GRANDI OBESI SOTTOPOSTI A BY-PASS DIGIUNO-ILEALE, estratto da Minerva Chirurgica, Vol. 39, N. 20, 1984

[5] W. Montorsi, LA SINDROME DELL'INTESTINO CORTO, Archivio ed Atti della Società Italiana di Chirurgia, LXXXV Congresso, 1983

Questo lavoro è stato presentato al 6° Corso Convegno Internazionale Radiodiagnostica e Terapie Integrate in Oncologia (Roma 17-22 feb. 1986) sul Suppl. n°1 del vol. 11/1 di RAYS, ed è qui pubblicato per gentile concessione del Direttore del Corso prof. A. Romanini.

APPLICAZIONE DEI SISTEMI ESPERTI NELL'AUTOMAZIONE D'UFFICIO: IL PROTOTIPO CHAOS

Raffaela Vasallo, Annamaria Zanaboni

Dipartimento di Scienze dell'Informazione - Università di Milano

RIASSUNTO

Viene qui presentato un sistema per l'automazione d'ufficio, CHAOS (Commitments Handling Advanced Office System), che è in grado di gestire conversazioni tra i membri di una organizzazione. Il sistema è basato sulla metafora del gioco, e considera i suoi utenti come un insieme di giocatori che giocano un gioco naturale, nel quale le loro mosse cambiano le Regole (Costitutive) del gioco stesso. CHAOS mantiene una rappresentazione delle Regole del gioco, e può "guidare" ogni utente ad agire nel proprio insieme di possibilità.

ABSTRACT

An office automation system, CHAOS (Commitments Handling Advanced Office System), is presented which is able to handle conversations between the members of an organisation.
The system is based on the game metaphor, and considers its users as a set of players playing a natural game, in which their moves change the (constitutive) rules of the game itself.
CHAOS mantains a representation of the game rules, and can "guide" each user to act inside his/her set of possibilities.

1. INTRODUZIONE

CHAOS (Commitments Handling Advanced Office System) è la realizzazione prototipale di un sistema di gestione delle conversazioni in ambito Office Automation, in grado di costruire una rappresentazione dell'organizzazione in cui è inserito e apprendendo dalle conversazioni che gestisce di modificare la sua rappresentazione dell'organizzazione e l'insieme di Atti Linguistici che rende possibili ai suoi utenti.

CHAOS ha costituito il nucleo centrale delle tesi di Laurea in Scienze dell'Informazione della autrici dell'articolo, [24] [25] che esse hanno svolto all'interno di un gruppo di ricerca che si occupa di problemi di Automazione d'Ufficio di cui fanno parte G. De Michelis, F. De Cindio, C. Simone, L. Pomello.

L'approccio, qui presentato, al problema dell'Automazione d'Ufficio non si basa su una descrizione strutturale o funzionale del lavoro d'ufficio, nè guarda al flusso delle informazioni, alle procedure svolte o all'aspetto decisionale nelle attività.
L'ufficio è invece considerato una componente di un sistema complesso, di una organizzazione sociale, prodotto dell'attività umana.

Si è assunto un approccio relazionale per caratterizzare le organizzazioni come sistemi e il punto di partenza dell'analisi svolta è la Teoria dei Sistemi Autopoietici, come teoria dei sistemi viventi, di H. Maturana e F. Varela [17].

Per costruire il modello di un organizzazione vanno individuate le Regole Costitutive, [7] [8], che creano lo spazio di possibilità, sul piano della comunicazione interpersonale, in cui agiscono i suoi membri per realizzare le relazioni che permettono all'organizzazione di mantenere la propria identità a fronte delle perturbazioni esterne.
Gli uomini, anche come membri di una organizzazione, vivono nel linguaggio e sono proprio i processi comunicativi che concorrono a definire ciò che è co-

stitutivo di un'organizzazione. Le Regole Costitutive di una organizzazione definiscono perciò lo spazio degli Atti Linguistici che sono possibili ai suoi membri e i protocolli comunicativi che essi seguono.

L'attenzione è quindi rivolta all'analisi della pragmatica della comunicazione secondo la Teoria degli Atti Linguistici di J. Searle, [19] [20], e alle relazioni tra parole e mondo secondo la Teoria delle Competenze Comunicative di F. Flores [13].
Dal punto di vista pragmatico, in un ufficio le persone non comunicano "trasferendo informazioni" ma creando, tenendo conto di, iniziando nuovi impegni.

Il disegno di nuovi strumenti per l'ufficio può essere fatto sulla base di questo approccio ed un primo esempio è costituito dal programma COORDINATOR, realizzato dalla Action Technologies e disegnato da F. Flores, che gestisce la rete di conversazioni dell'ufficio [1].

Questo articolo descrive un prototipo COORDINATOR di II generazione, intendendo con tale nome un sistema capace di apprendere delle conversioni che gestisce e di modificare in base a ciò le possibilità comunicative che offre ai suoi utenti.

Il prototipo CHAOS (Commitments Handling Advanced Office System) è stato implementato sulla base della formalizzazione delle Regole Costitutive che definiscono una semplice struttura organizzativa. Sono stati costruiti i modelli delle conversazioni che caratterizzano una *cooperativa,* utilizzando il linguaggio delle reti SA (Automi Sovrapposti), una sottoclasse delle reti di Petri [6].
Le reti delle conversazioni definiscono i requisiti e le specifiche del sistema, implementato in LISP utilizzando tecniche di Artificial Intelligence.

Il sistema CHAOS è in grado di

- gestire le conversazioni tenendo conto delle relazioni tra gli Atti Linguistici che occorrono in esse. Ad esempio: l'invio di una offerta e la conseguente attesa di una risposta: la ricezione di una controfferta che modifica alcune delle condizioni di soddisfazione di una precedente richiesta: l'invio di una promessa che causa l'assunzione di un nuovo impegno;

- consentire un controllo delle relazioni temporali tra gli Atti Linguistici e gli impegni assunti con la possibilità di leggere un "orologio" e verificare ritardi e scadenze;

- costruire modelli della struttura organizzativa dal punto di vista delle competenze dei membri circa le commesse su cui sono impegnati, i clienti con cui sono in contatto e conoscenze acquisite tramite l'esperienza;

- permettere l'esecuzione di Atti Linguistici parzialmente specificati, nei quali il partner non esplicitato viene identificato in base ai modelli di cui sopra;

- costruire un semplice modello del lessico della cooperativa, permettendo agli utenti di identificare le proprie conversazioni tramite dei sinonimi;

- incorporare alcune delle Regole Costitutive della cooperativa grazie alla capacità di distinzione degli Atti Linguistici che causano una riorganizzazione delle relazioni sociali tra i suoi membri.
Sulla base di questa conoscenza, il sistema modifica la propria interfaccia utente aggiornando lo spazio di possibilità degli utenti di aprire conversazioni.

2. CONSIDERAZIONI PRELIMINARI

Una conversazione avviene tra due interlocutori, uno dei quali è colui che la apre.
Chiameremo *attore* l'individuo che apre la conversazione e, *partner* l'altro interlocutore.
Per semplicità abbiamo considerato solo il caso di conversazioni aperte con un atto commissivo o direttivo (e per una organizzazione cooperativa questa non è una grossa restrizione: nessun membro ha infatti l'autorità per fare delle dichiarazioni che possono essere percepite dall'organizzazione; e di solito gli atti assertivi sono eseguiti in seguito ad una richiesta esplicita).
Per atto linguistico commissivo si intende, in base all'analisi di S. Searle [19] [20], un atto linguistico il cui contenuto illocutivo è un'offerta, una promessa, ecc...; per atto linquistico direttivo s'intende un atto linguistico il cui contenuto illocutivo è una richiesta, un ordine, una preghiera, ecc.
Abbiamo quindi ristretto l'analisi alle conversazioni per l'azione, trascurando quelle per la possibilità (che invece sono molto frequenti in un caso reale, anche per le organizzazioni cooperative) [13] [16] [11].

In una conversazione viene preso un impegno da entrambi gli interlocutori. In particolare nelle conversazioni per l'azione aperte da un atto commissivo o direttivo, uno dei due si impegna a "fare" qualcosa, l'altro a garantire il mantenimento delle condizioni (quelle che dipendono da lui) perchè il primo porti a termine il suo lavoro. Chiameremo *attivo* l'individuo che si impegna a "fare", e *passivo* l'altro.
Ad esempio, se A chiede a B di fare per lui un certo lavoro entro il giorno seguente, e se B accetta, allora in questa conversazione A è attore passivo e B è partner attivo; viceversa, se A offre a B di collaborare con lui (con B) sul progetto che B ha in corso, allora A è attore attivo e B partner passivo.

In una qualsiasi conversazione si possono identificare due fasi di rispetto all'impegno: durante la fase iniziale i due interlocutori si accordano sull'entità dell'im-

pegno (ad esempio, se A chiede a B di fare una certa cosa, allora B può rispondere che potrebbe fare un sottoinsieme di ciò che gli è stato richiesto, poi A può proporre ancora un'altra soluzione, e così via), e quando (e se) l'accordo viene raggiunto, si passa alla seconda fase, in cui l'impegno viene soddisfatto, ma anche può essere ridefinito o revocato.
Naturalmente si può subito declinare l'offerta o la richiesta, e in generale è possibile chiudere la conversazione senza portare a termine nè revocare l'impegno (ad esempio perchè sono venute meno le condizioni che rendevano utile e conveniente l'assunzione di quell'impegno; condizioni che possono non dipendere dalla volontà dei singoli). In tal caso diciamo che la conversazione è stata cancellata.

3. LE REGOLE DEL GIOCO

Le regole che regolano i giochi linguistici, si apprendono giocando. Gli esseri umani, attori ma anche osservatori del gioco, agendo sperimentano ciò che è permesso o non permesso nel gioco (gli uomini sono *"gettati nel mondo"*) [26].

Come descritto in [7], seguendo la caratterizzazione di Searle, le regole che regolano il comportamento di *n* attori in un gioco possono essere suddivise in Costitutive e Prescrittive (queste ultime sono dette regolative da Searle).
Le prime generano (definiscono) lo spazio di possibilità di ciascun attore, le seconde selezionano particolari comportamenti tra tutti quelli definiti dalle Regole Costitutive.
Seguire una strategia predefinita, significa seguire una sequenza di Regole Prescrittive.

Pertanto, ciò che interessa nella costruzione del modello di un'organizzazione è l'individuazione delle Regole Costitutive della comunicazione che i suoi membri seguono, perchè esse definiscono lo spazio di possibilità entro il quale essi si muovono per realizzare la relazione che permette la coesione dell'organizzazione, il mantenimento della sua identità. «È come se dovessimo "guardare attraverso" i fenomeni; la nostra ricerca non si rivolge però ai fenomeni, ma si potrebbe dire alle "possibilità" dei fenomeni» [23], pag. 60.

3.1 Le Regole Costitutive

Le *Regole Costitutive* possono essere suddivise in due classi:

1. regole che creano il dominio di possibilità dei membri intesi come soggetti sociali: sono quelle che permettono l'interazione tra esseri umani, che caratterizzano gli esseri umani come soggetti sociali.
2. regole che creano il dominio delle possibilità dei membri all'interno di una particolare organizzazione. Sono diverse nelle diverse organizzazioni: esse infatti caratterizzano ogni organizzazione.

   In [7] viene data una esemplificazione di Regole Costitutive della seconda classe nelle organizzazioni economiche:

   - regole che definiscono quali sono i membri dell'organizzazione
   - regole che definiscono i ruoli dei membri dell'organizzazione
   - regole che definiscono i protocolli di interazione tra i membri dell'organizzazione.

Le Regole Costitutive definiscono lo spazio percettivo dell'organizzazione, e quelle della seconda classe possono cambiare in seguito all'interazione stessa dei membri dell'organizzazione: la relazione tra i membri dell'organizzazione viene ridefinita in seguito alle loro mosse nel loro spazio di possibilità.
Essendo la definizione di una possibilità, una Regola Costitutiva non può venire violata (semplicemente può non venire eseguita).

Una particolare organizzazione è caratterizzata da un insieme di regole del primo tipo (quelle che governano le conversazioni tra esseri umani), e da un insieme di regole del secondo tipo, ad essa peculiari.
Nell'organizzazione considerata, le regole del primo tipo che abbiamo preso in considerazione sono quelle che governano le conversazioni aperte da una promessa o da una richiesta.

Le regole del secondo tipo caratterizzano le competenze dei membri dell'organizzazione.
Queste ultime si generano in seguito all'esecuzione di regole del primo tipo, alterando lo spazio di possibilità di ogni individuo all'interno delle conversazioni in cui è coinvolto, ed i possibili atti linguistici da lui eseguibili (che possono cioè essere percepiti dall'organizzazione, e che non causano la sua disintegrazione).

La competenza di un socio ha differenti aspetti: è competenza su un *argomento*, su un *cliente* (il socio conosce bene un certo cliente, ha già avuto contatti con lui), su una *commessa* (il socio sta lavorando su certe parti di una data commessa), su una specifica *azione,* che è il lavoro che egli sta eseguendo (e questo aspetto caratterizza in modo univoco l'impegno che un socio attivo in una conversazione si assume).

Abbiamo modellato lo spazio di possibilità dei membri dell'organizzazione presa in considerazione (cioè le Regole Costitutive seguite dagli individui all'interno dell'organizzazione) tramite il linguaggio delle reti degli automi sovrapposti (reti SA, [6]).

Le regole da noi individuate ed analizzate sono quel-

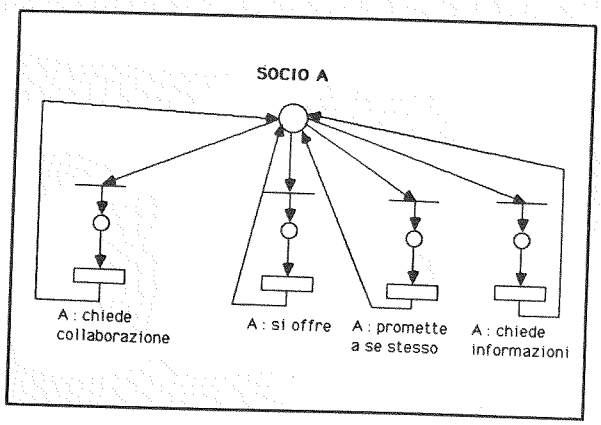


Fig. 1

le che governano le conversazioni aperte con una offerta, con una richiesta (di informazione o di collaborazione), e con una promessa a se stessi.

Nella figura 1 viene mostrato lo spazio di possibilità di ciascun socio per comunicare con gli altri soci.

Un esempio di regola è dato dalle reti di figura 2 e figura 3 (una per l'attore e una per il partner, che per chiarezza non abbiamo sovrapposto), che modellano lo spazio di possibilità all'interno del quale si muovono i due interlocutori di una conversazione aperta con una richiesta di collaborazione.

A chiede a B di impegnarsi a collaborare con lui su qualche progetto.

A specifica un termine entro il quale si attende una risposta da B, ed un termine entro il quale si attende che B completi il lavoro.

In seguito al ricevimento di una richiesta, B può:

- declinare (con la seguente chiusura della conversazione);
- accettare (nel qual caso B risulta impegnato);
- controffrire: può cioè proporre una alternativa alla richiesta ricevuta (rispetto alla scadenza temporale o dell'entità dell'impegno).

Ad esempio:

A: (apertura di conversazione con una Richiesta di collaborazione)
argomento: funzioni di I/O
cliente: XXX
commessa: graUNIX
azione: Tu che sai tutto sull'I/O, perchè non dai un'occhiata al modulo di interfaccia con la stampante di questo sistema su cui ci stiamo dannando da tre mesi?
Tieni presente che dobbiamo presentare una versione funzionante (o quasi) per il mese di febbraio. Fammi sapere entro la settimana.

B: (controfferta sull'azione)
azione: Non so se lo sai (ma dovresti saperlo...), per fare quello che mi chiedi devo andare a studiare le funzioni di I/O della stampante, che non è cosa da poco. Se chiedi aiuto anche a qualcuno (uno a caso: Andrea) che si intenda della stampante, più che volentieri: io guardo l'aspetto "invio comandi", lui quello "ricezione/esecuzione comandi".

In seguito alla controfferta, B può aspettarsi un declino, una accettazione (nel qual caso risulterà automaticamente impegnato nei termini concordati), o l'inizio di una contrattazione.

Dopo la presa dell'impegno, B può:

- fare rapporto, avendo portato a termine il suo impegno;
- revocare: questa revoca, però, fatta da un partner attivo (e non da un attore attivo), non porta immediatamente alla chiusura della conversazione. A può ancora controrichiedere, e B può accettare o declinare; nel caso dell'accettazione B sarà impegnato in termini differenti rispetto a prima, nel caso del declino i due interlocutori possono continuare a contrattare per raggiungere un accordo. È l'attore, comunque, ad avere la possibilità di chiudere la conversazione, tramite una cancellazione.

- ricevere una controrichiesta: A può chiedere di cambiare i termini dell'impegno, originando una contrattazione, o ricevendo un declino o una accettazione come descritto a proposito della controrichiesta conseguente ad una revoca.

4. DESCRIZIONE DEL SISTEMA

Il sistema CHAOS:

1 - è un supporto utile per la conversazione tra due o più interlocutori;

2 - implementa il modello dell'organizzazione come rete di impegni, ed è in grado di estrarre dalle conversazioni che hanno luogo tra gli utenti la conoscenza necessaria per tenere traccia della rete degli impegni in cui essi sono coinvolti:

3 - conosce il lessico dell'organizzazione che può essere strutturato in modi sempre più complessi in future espansioni del sistema.

4.1 Nucleo concettuale

Il sistema contiene:

1 - istanze delle conversazioni attive nell'organizzazione in un dato istante, modellate in reti SA;

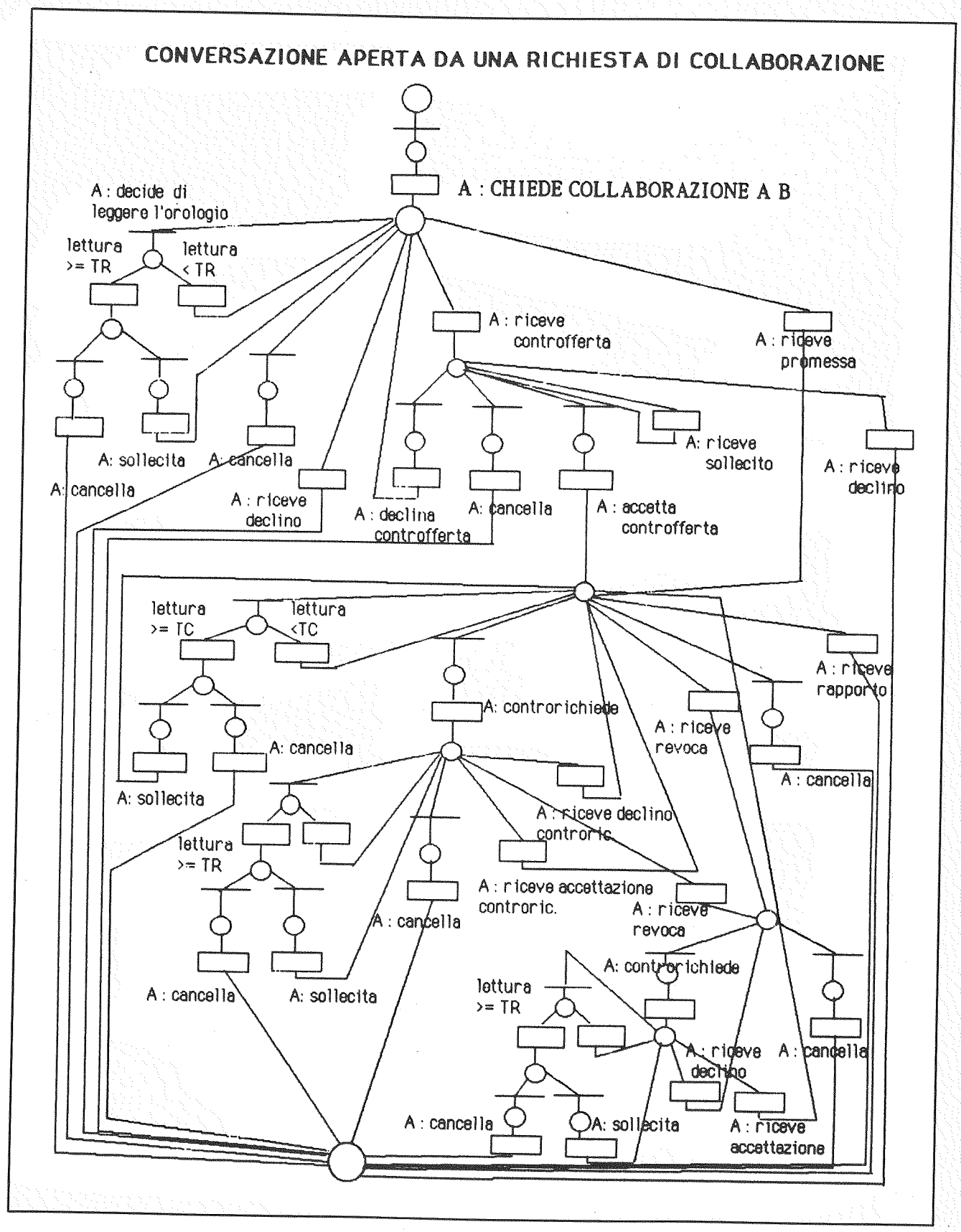

CONVERSAZIONE APERTA DA UNA RICHIESTA DI COLLABORAZIONE
A : CHIEDE COLLABORAZIONE A B
A : decide di leggere l'orologio
lettura >= TR
lettura < TR
A: sollecita
A: cancella
A: cancella
A : riceve controfferta
A : riceve promessa
A: riceve sollecito
A : riceve declino
A : riceve declino
A : declina controfferta
A: cancella
A : accetta controfferta
lettura >= TC
lettura <TC
A: controrichiede
A : riceve rapporto
A : riceve revoca
A: cancella
A: cancella
A: sollecita
A : riceve declino controric.
lettura >= TR
A : riceve accettazione controric.
A : riceve revoca
A : cancella
A: controrichiede
A : cancella
A: sollecita
lettura >= TR
A : riceve declino
A : cancella
A : cancella
A: sollecita
A : riceve accettazione

Fig. 2

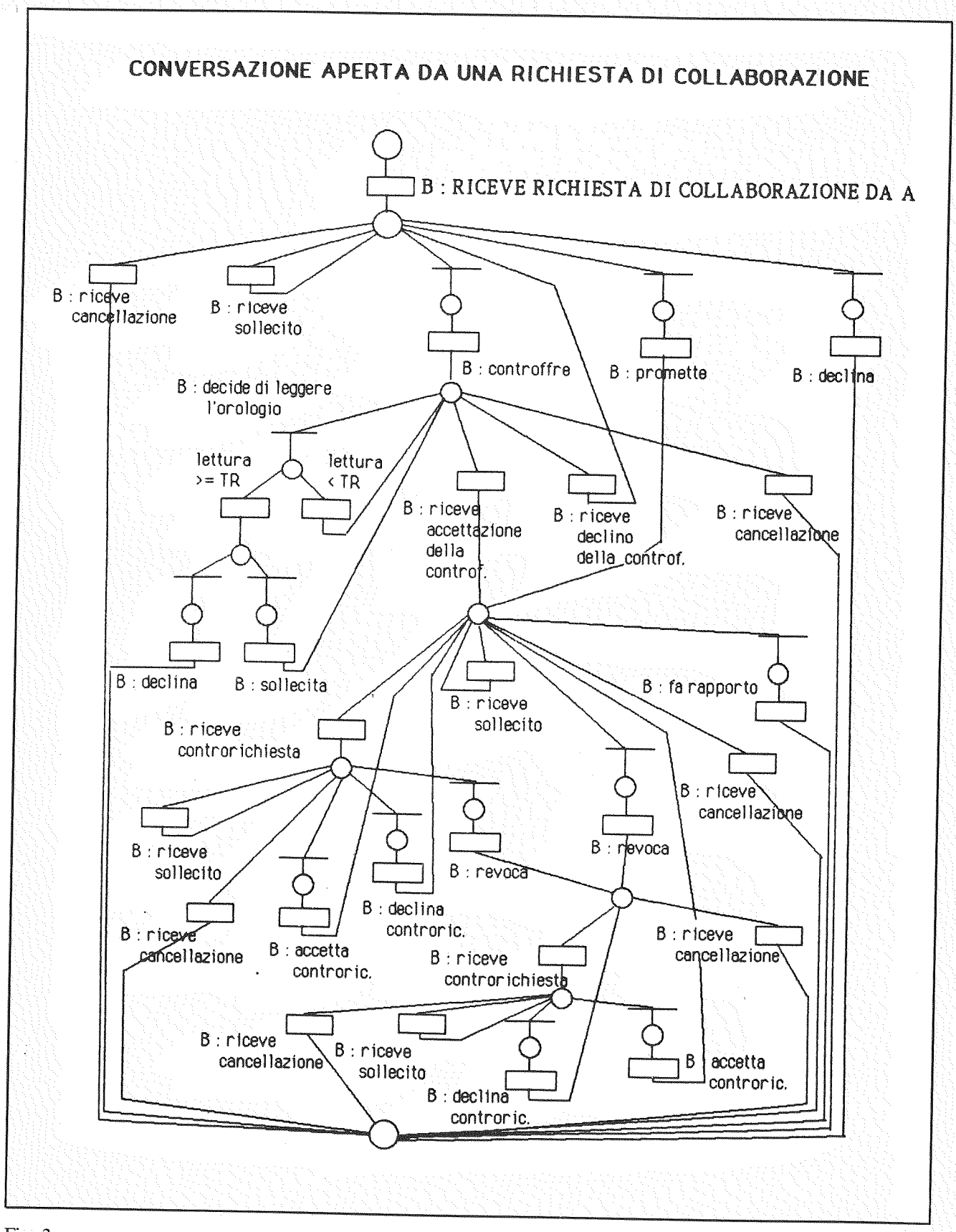

CONVERSAZIONE APERTA DA UNA RICHIESTA DI COLLABORAZIONE
B : RICEVE RICHIESTA DI COLLABORAZIONE DA A
B : riceve cancellazione
B : riceve sollecito
B : controffre
B : promette
B : declina
B : decide di leggere l'orologio
lettura >= TR
lettura < TR
B : riceve accettazione della contrf.
B : riceve declino della contrf.
B : riceve cancellazione
B : declina
B : sollecita
B : riceve controrichiesta
B : riceve sollecito
B : fa rapporto
B : riceve cancellazione
B : riceve sollecito
B : revoca
B : revoca
B : riceve cancellazione
B : declina controric.
B : accetta controric.
B : riceve controrichiesta
B : riceve cancellazione
B : riceve cancellazione
B : riceve sollecito
B : accetta controric.
B : declina controric.

Fig. 3

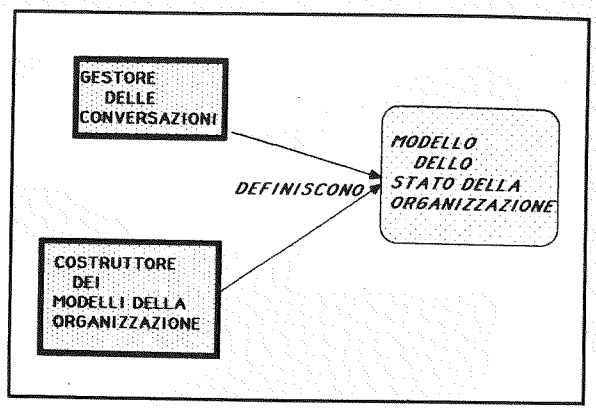


Fig. 4

2 - un gestore di conversazioni che sposta le marche sulle reti delle conversazioni, crea nuove istanze di reti, e le mantiene aggiornate;

3 - un costruttore del modello dell'organizzazione, cioè della rete degli impegni, che genera lo spazio delle possibilità aperto ad ogni utente in ogni istante.

Nella figura 4 viene mostrato il nucleo concettuale del sistema.

4.2 Moduli software

Affinchè il sistema sia utilizzabile, è necessario che il nucleo venga avvolto da un involucro che permetta la comunicazione utenti-kernel: l'interfaccia con l'utente.

I moduli software che possiamo individuare ad altissimo livello sono quattro:

- *interfaccia*
- *gestore delle conversazioni*
- *sistema esperto*
- *lessico*

con le seguenti caratteristiche e relazioni, che sono mostrate nella figura 5.

1) - L'interfaccia *(colloquia con l'utente)*

- Fornisce delle funzioni di esplorazione della rete di conversazioni di un dato utente;
- È in grado di permettere all'utente di eseguire atti linguistici, anche parzialmente specificati;
- Conosce le reti dei diversi tipi di conversazione ed in base ad essa sa ciò che ogni membro può fare in ogni istante all'interno di una conversazione.

È modificata dal modulo esperto: la sua capacità di permettere l'esecuzione di atti linguistici parzialmente specificati in base alla configurazione della rete degli impegni in un dato momento dipende dalla capacità di apprendimento del modulo esperto.

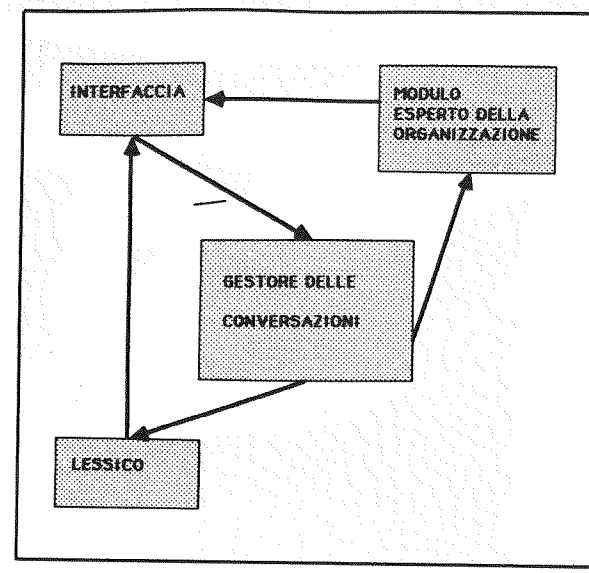


Fig. 5

2) - Il gestore *(delle conversazioni mantiene aggiornato l'insieme delle conversazioni).*

È attivato dall'interfaccia ogni volta che un utente esegue un atto linguistico.

Il gestore ad ogni esecuzione di atto linguistico:

- chiama, se necessario, il modulo esperto che ha la capacità di distinzione sulle regole;
- chiama il modulo lessicale che estrae dal contenuto proposizionale dell'atto le informazioni necessarie per aggiornare la struttura del lessico dell'organizzazione.

3) - Il modulo esperto *(mantiene aggiornato il modello dell'organizzazione come rete di impegni, e modifica l'interfaccia a seconda dello spazio di possibilità di ciascun utente (che dipende in ogni istante dalla configurazione della rete degli impegni vigente in quell'istante nell'organizzazione))*

Questo è l'unico modulo in grado di distinguere gli effetti dell'esecuzione di ogni atto linguistico sulla rete degli impegni.

È attivato:

- dal gestore delle conversazioni ogni volta che viene eseguito un atto linguistico: in questo caso il modulo esperto crea o distrugge le regole relative all'impegno sotteso dalla conversazione di cui l'atto linguistico è parte;

- dall'interfaccia:
  a) ogni volta che viene aperta una conversazione

senza che l'attore abbia specificato il partner: in questo caso il modulo esperto esamina la configurazione della rete degli impegni tra i membri dell'organizzazione, ed eventualmente fornisce un interlocutore che sia competente sulle competenze richieste dall'attore;

b) ogni volta che viene aperta una conversazione con una richiesta di informazione: in questo caso il modulo esperto stabilisce se nello spazio di possibilità del partner c'è la possibilità di declinare.

4) - il modulo lessicale *(interagisce col gestore e con l'interfaccia, prendendo dal primo nuove conoscenze sul lessico dell'organizzazione e passandole alla seconda per aumentare la sua capacità di modifica e adattamento alla struttura dell'organizzazione)*

5. SPECIFICHE

Si è fatta la scelta implementativa del sistema ad oggetti, in cui gli oggetti sono le conversazioni.

Oggetti

Sull'oggetto conversazione sono eseguibili delle operazioni nel modo più ortogonale possibile (stampa, modifica, ecc...).

Inoltre l'universo delle conversazioni può essere visto sotto diversi punti di vista (view), intendendo come punto di vista il valere di una certa proprietà sull'insieme delle conversazioni.

Utilizzando i punti di vista, si possono selezionare sottoinsiemi di conversazioni, che possono ancora essere guardate sotto dei punti di vista, e così via.

Operazioni e punti di vista sono a loro volta applicabili in modo ortogonale.

Ecco nella figura 6 cos'è per noi un oggetto conversazione:

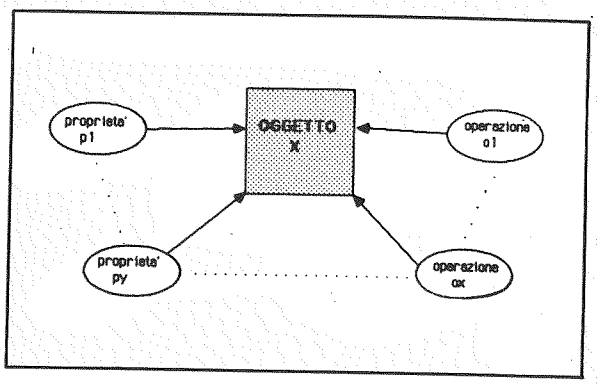


Fig. 6 Gli oggetti conversazione.

Lessico

Il sistema conosce il lessico dell'organizzazione.
Intendiamo per lessico il linguaggio parlato dai membri dell'organizzazione, insieme con la sua struttura semantica.

Eseguire un atto linguistico significa prendere un impegno.
L'impegno include delle condizioni di soddisfazione e un contenuto proposizionale.
Una conversazione è una sequenza temporale di atti linguistici.

Abbiamo distinto i due interlocutori in attivo e passivo, a seconda che si assumano l'impegno di fare qualcosa o di creare e mantenere le condizioni per cui l'altro possa portare a termine il suo impegno.
Gli impegni particolari definiti nelle conversazioni vengono distinti dal nostro sistema in base al contenuto proposizionale.
Se la conversazione *c1* è sull'azione *a1*, allora diremo che *a1* è l'identificatore univoco di *c1*, e *a1* è un elemento del lessico dell'organizzazione.

Se tra i due interlocutori c'è stata una fase di contrattazione, tutte le azioni che sono entrate in gioco sono ugualmente identificatori univoci della conversazione, e vengono considerate dal sistema come sinonimi lessicali dell'azione considerata.
Ciò è sensato, perchè quando A e B stanno colloquiando in una conversazione, se A propone l'azione *a1* e B propone come sua alternativa l'azione *a2*, ciò significa che per B *a2* in qualche modo è collegata con *a1*, o perchè specifica meglio i termini dell'impegno, o perchè è più generale, o perchè è perfettamente equivalente ma si usa di più, ecc...
Questa relazione tra *a1* e *a2* è chiara per B (e possibilmente anche per A e per gli altri membri dell'organizzazione), ma può non esserlo per individui esterni all'organizzazione.
In questo senso i sinonimi sono tali all'interno dell'organizzazione, ma possono non esserlo in un altro contesto, e per questo noi parlamo il *lessico* dell'organizzazione e non *di linguaggio*.

Il lessico dell'organizzazione e l'insieme delle azioni correlate tramite la relazione di sinonimo.

Con il lessico pertanto si struttura l'universo delle conversazioni: una conversazione può essere "chiamata" in tanti modi.
Questa strutturazione è da noi chiamata semantica, anche se per il movimento non viene fatta alcuna analisi semantica del contenuto proporzionale degli atti linguistici.
Nel futuro si potrà arricchire l'intelligenza di questo modulo, introducendo analisi semantica sull'azione e interazioni con l'utente per chiarimenti e spiegazioni.

Modello dell'organizzazione

Il modello dell'organizzazione è costituito dalla rete di impegni tra i suoi membri.
Gli impegni considerati in questo modello sono quelli sulle aree di competenza, sui clienti, sulle commesse.
Se il membro A dell'organizzazione è coinvolto in una conversazione in cui egli ha preso l'impegno sull'area di competenza *ar,* sul cliente *cl,* sulla commessa *cm,* allora il sistema ha appreso che A è competente sull'area *ar,* conosce il cliente *cl* e sta lavorando sulla commessa *cm.*

In conclusione, è possibile selezionare oggetti nell'universo delle conversazioni secondo:
- LE PROPRIETÀ
- IL CONTENUTO PROPOSIZIONALE DELL'IMPEGNO SOTTESO DA OGNI CONVERSAZIONE
- L'OGGETTO (AREA, CLIENTE, COMMESSA)

Avendo fatto la scelta di sistema ad oggetti, ad un livello molto alto di disegno, si può schematizzare quanto detto nella figura 7.

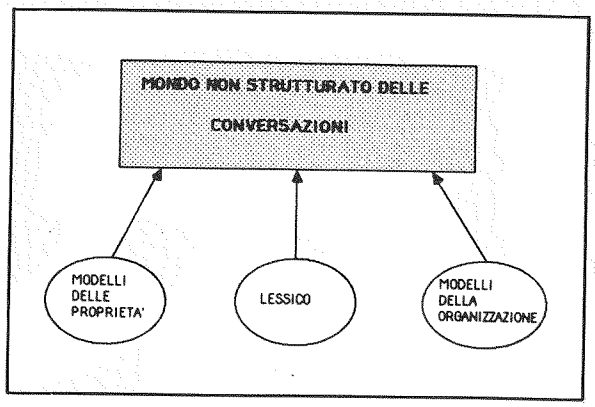


Fig. 7

6. ARCHITETTURA DEL SISTEMA

La nostra implementazione simula una rete di personal, in cui la comunicazione tra gli utenti è asincrona

Ogni utente ha a disposizione:
1 - un *data base*
2 - un *buffer* per deposito temporaneo dei messaggi ricevuti
- un paio di *occhiali da conversazione*
- un'*interfaccia*

I modelli delle proprietà, il modello dell'organizzazione e il lessico sono comuni a tutti.

1) Nel *data base* di ogni utente ci sono le strutture dati relative a tutte le conversazioni in cui egli è coinvolto.
   L'insieme delle conversazioni è pertanto distribuito nei vari data base personali.
   Poichè l'invio e la ricezione di un messaggio non avvengono nello stesso istante, è possibile che in un certo momento la struttura dati di una certa conversazione *c* nel data base dell'utente *i* sia diversa da quella della stessa *c* nel data base dell'utente j, interlocutore di *i* in *c.*
   Quando l'utente *j* vedrà i messaggi arrivati sulla conversazione *c,* le due strutture ritorneranno a coincidere.

2) Il *buffer* è un recipiente in cui vengono depositati i messaggi arrivati sulle conversazioni dell'utente. Essi scompaiono dal buffer non appena l'utente li ha letti e la struttura dati della conversazione sia stata aggiornata.

3) Il paio di *occhiali* consiste in una strutturazione dell'insieme delle proprie conversazioni secondo le proprietà già descritte. Quindi l'insieme dei modelli degli attributi è stato distribuito negli occhiali di ogni utente.
   I modelli delle proprietà che definiscono i punti di vista vengono aggiornati in seguito all'esecuzione di certi atti linguistici, o in dipendenza dallo scorrere del tempo.
   Indirettamente, tali modelli influenzano l'interfaccia, perchè le possibilità che essa offre (e qui precisamente l'insieme di conversazioni su cui operare) dipendono dallo stato dei modelli della proprietà in un certo istante.

4) Ogni utente ha la percezione di una *interfaccia* personalizzata, perchè il modulo di interfaccia lo sa guidare conoscendo lo stato dell'oggetto si cui egli ha chiesto di operare: l'interfaccia sa in ogni momento in una data conversazione lo spazio di possibilità dell'utente.
   Inoltre l'interfaccia si modifica seguendo l'evoluzione dell'organizzazione: permette l'esecuzione di atti linguistici senza un interlocutore specifico, coinvolgendo automaticamente nella conversazione un individuo che sia competente.
   Essa è anche influenzata dai modelli delle proprietà, come già si è detto.
   Il modulo di interfaccia contiene:
   - le regole dell'organizzazione
   - un data base per contenere le maschere di lettura e stampa
   - funzioni di colloquio con l'utente
   - funzioni di ispezione delle regole dell'organizzazione.

È invece centralizzato:
1 - il *modulo esperto*
2 - il *gestore delle conversazioni*
3 - il *lessico*

1) il *modulo esperto* contiene:
- il modello dell'organizzazione

- alcune entry per la modifica dell'interfaccia e per l'aggiornamento del modello.

2) Il *gestore* contiene:
- la descrizione di ogni utente (data base, occhiali, buffer)
- funzioni di basso livello di accesso e modifica delle conversazioni
- funzioni associate ai vari atti linguistici: lo scatto di una transizione in una rete equivale all'esecuzione di una funzione che può implicare:

    - colloquio con l'utente
    - aggiornamento dei modelli delle proprietà
    - chiamata al modulo esperto
    - aggiornamento del lessico,

  e sempre fa muovere le marche sulla rete della conversazione.
  Poichè la comunicazione è asincrona, una transizione di comunicazione non si considera scattata finchè l'atto linguistico non è stato sia inviato che ricevuto. Pertanto le transizioni di comunicazione hanno associate due funzioni: una da eseguire in invio, l'altra in ricezione.
- funzioni di mantenimento dei modelli delle proprietà (strutturazione del mondo delle conversazioni)
- funzioni di gestione (invio e ricezione) dei messaggi.

3) Il *lessico* contiene:
- la struttura lessicale dell'organizzazione (descritta dalla relazione di sinonimo)
- alcune entry di aggiornamento e ispezione di tale struttura.

Questo modulo è in diretto contatto col gestore, ma non ha un legame con il modulo esperto. Ulteriori sviluppi del prototipo potranno aggiungere complessità al modulo lessicale e a quello esperto.
Ed ecco la figura 8 che mostra le relazioni tra ciò che è centralizzato e ciò che non lo è in riferimento a due utenti, *i,* e *j,* e la figura 9 che mostra la configurazione del sistema.

7. IL SISTEMA CHAOS

Il sistema è stato implementato sotto il sistema operativo *Unix*, in ambiente *Franz Lisp,* [15] utilizzando un package (PEARL) [10] [12] adatto all'implementazione di programmi di intelligenza artificiale.
PEARL è molto potente: facilita la programmazione ad oggetti, e mette a disposizione:
- funzioni per la creazione di data bases di oggetti,
- funzioni per il riempimento di oggetti non solo tramite il pattern-matching, ma anche tramite unificazione,
- meccanismi per indicizzare gli oggetti anche con chiavi multiple
- funzioni per accedere alle singole componenti degli oggetti in un tempo costante, indipendentemente dal numero dei componenti,
- ed altre ancora.

Il sistema ospite è il *VAX-11/750* del Centro di Calcolo automatico dell'Università degli Studi di Milano.

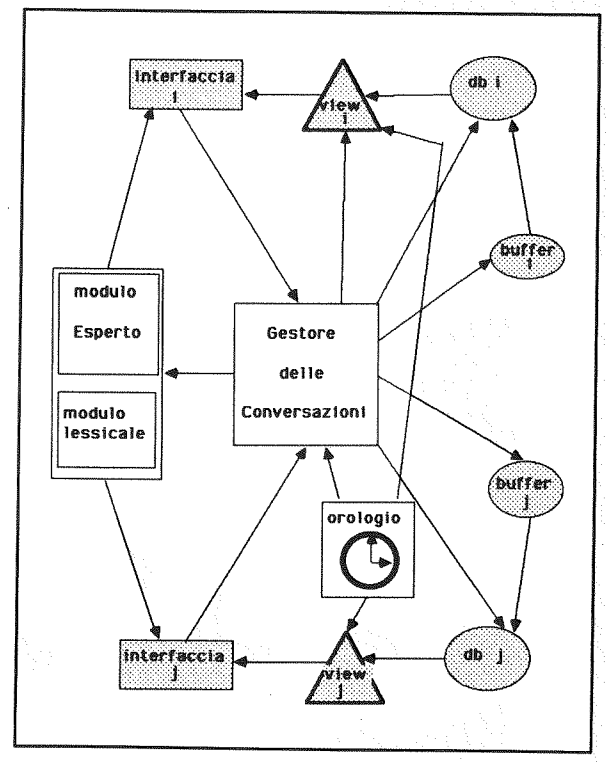


Fig. 8

8. BIBLIOGRAFIA

[1] Action Technologies, COORDINATOR-USES MANUAL AND REPORT, 1984

[2] J. Allen, ANATOMY OF LISP, Mac Graw-Hill, New York, 1978

[3] C.V. Bullet, J.L. Bennet, OFFICE WORKSTATION USE BY ADMINISTRATIVE MANAGERS AND PROFESSIONALS, CISR WR # 84, Sloan School of Management, MIT, 1983

[4] G. Chili, LINGUAGGIO E LAVORO D'UFFICIO: PROSPETTIVE D'INDAGINE, in "Office Automation: Metodi e tecnologie", G. Degli Antoni, G. Occhini (eds), Collana AICA Informatica, Masson Italia Editori, Milano, 1986

[5] E. Charniak, C.K. Riesbeck, D. McDermott, ARTIFICIAL INTELLIGENCE PROGRAMMING, Laurence Erlbaum Associates Inc. Publishers, New Jersey, 1980

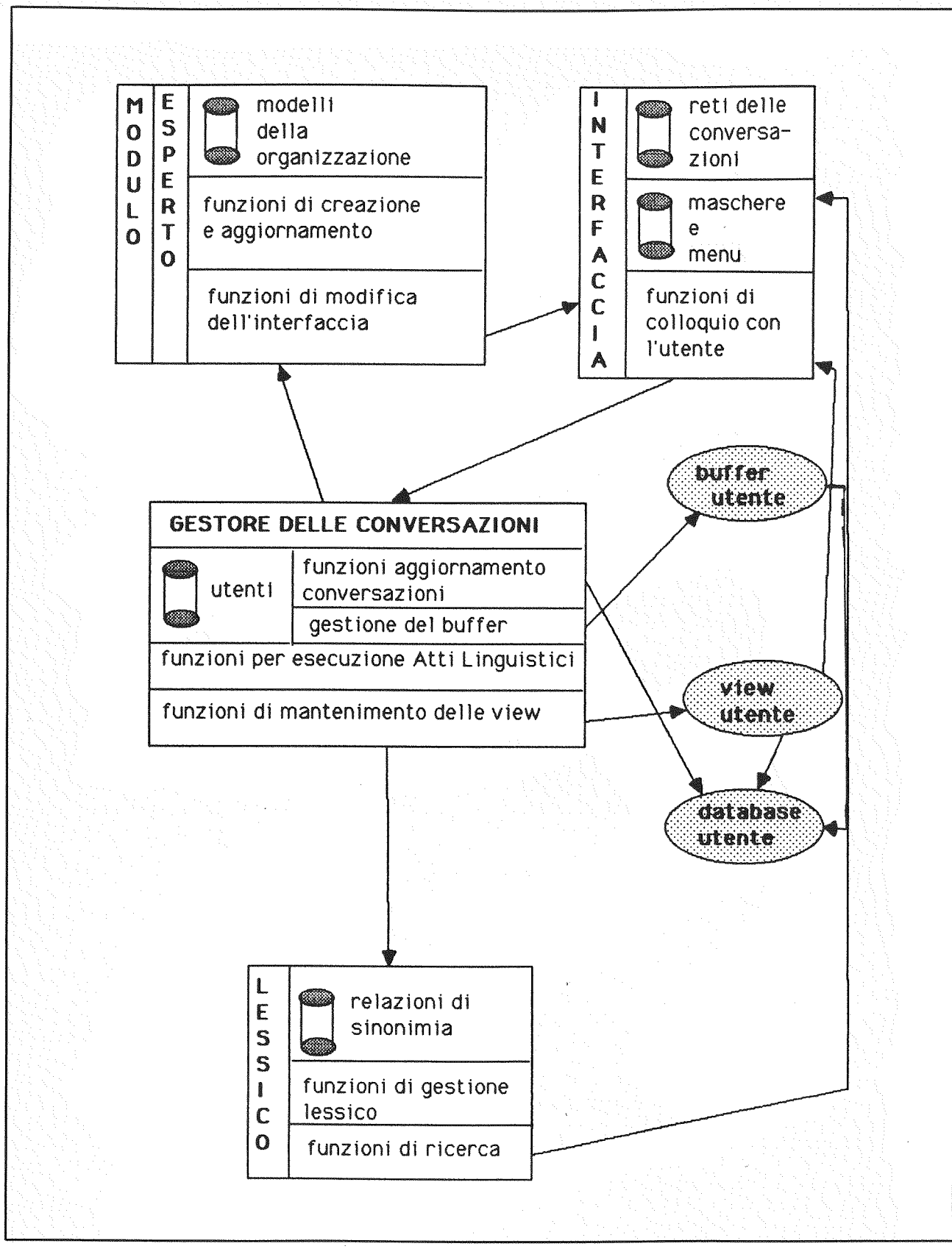

MODULO ESPERTO
modelli della organizzazione
funzioni di creazione e aggiornamento
funzioni di modifica dell'interfaccia
INTERFACCIA
reti delle conversa-zioni
maschere e menu
funzioni di colloquio con l'utente
buffer utente
GESTORE DELLE CONVERSAZIONI
utenti
funzioni aggiornamento conversazioni
gestione del buffer
funzioni per esecuzione Atti Linguistici
funzioni di mantenimento delle view
view utente
database utente
LESSICO
relazioni di sinonimia
funzioni di gestione lessico
funzioni di ricerca

Fig. 9

[6] F. De Cindio, G. De Michelis, L. Pomello, C. Simone, SUPERPOSED AUTOMATA NETS, in IFB 52, Springer Verlag, Berlin, 1982

[7] F. De Cindio, G. De Michelis, C. Simone, ORGANIZZAZIONE COME SISTEMA: QUALI RELAZIONI, QUALI MODELLI, Studi Organizzativi, n. 3-4, anno XV, 1984

[8] F. De Cindio, G. De Michelis, C. Simone, FORMAL COMPUTER SYSTEM IN SOCIAL ORGANIZATION, in Proc. Working Conference on Development and Use of Computer-based System and Tools, Aarhus, Denmark, 1985

[9] F. De Cindio, G. De Michelis, C. Simone, DIMENSIONI DELL'USABILITÀ, UNIVERSI DEL DISCORSO E SOFTWARE D'UFFICIO in "Usabilità del software", a cura di S. Bagnara (in corso di stampa).

[10] M. Deering, J. Faletti, R. Wilensky, USING THE PEARL AI PACKAGE (Package for Efficient Access to Representation in Lisp), Computer Science Div., EECS Dept., University of California, 1982

[11] R. Donà IMPLEMENTAZIONE DI SOFTWARE SU MODELLI CONVERSAZIONALI, Tesi di Laurea in Scienze dell'Informazione, Università degli Studi di Milano, aa. 1983-84

[12] J. Faletti, R. Wilensky, THE IMPLEMENTATION OF PEARL. A PACKAGE FOR EFFICIENT ACCESS TO REPRESENTATION IN LISP, Computer Science Div., EECS Dept., University of California, 1982

[13] F. Flores, J.J. Ludlow, DOING AND SPEAKING IN THE OFFICE, in "DDS: Issues and Challenges", F. Fisk, R. Sprague (eds), Pergamon Press, London, 1981

[14] F. Flores, T. Winograd, UNDERSTANDING COMPUTER AND COGNITION, Ablex, New York, 1985

[15] J.K. Foderaro, K.L. Sklower, THE FRANZ LISP MANUAL, in "Berkeley UNIX* Reference Manual", vol, 2c, Computer Science Div., EECS Depts., University of California, 1981

[16] T. Lacquaniti, MODELLI CONVERSAZIONALI NELL'UFFICIO DEL FUTURO, Tesi di Laurea in Scienze dell'Informazione, Università degli Studi di Milano, aa. 1983-84

[17] H. Maturana, F. Varela, AUTOPOIESI E COGNIZIONE. LA REALIZZAZIONE DEL VIVENTE, Marsilio Editori, Venezia, 1985

[18] C.A. Petri, KOMMUNICATIONSDIZIPLINEN, in Ansätze zur Organizationtheorie Rechnergestützer Informationssysteme, C.A. Petri (ed.), Munchen, Oldenburg, 1979

[19] J.R. Searle, ATTI LINGUISTICI. SAGGIO DI FILOSOFIA DEL LINGUAGGIO, Boringhieri, Torino, 1976

[20] J.R. Searle, PER UNA TASSONOMIA DEGLI ATTI ILLOCUTORI in "Gli atti linguistici. Aspetti e problemi di filosofia del linguaggio", Feltrinelli, Milano, 1978

[21] R. Wilensky, UNDERSTANDING GOAL-BASED STORIES, Technical Report 140, Computer Science Dept., Yale University, New Haven, CT, 1978

[22] R. Wilensky, META-PLANNING REPRESENTATION AND USING KNOWLEDGE ABOUT PLANNING IN PROBLEM SOLVING AND NATURAL LANGUAGE UNDERSTANDING, Cognitive Science, 5:3,1981

[23] L. Wittgenstein, RICERCHE FILOSOFICHE, Einaudi, Torino, 1967

[24] A. Zanaboni, UN SISTEMA PER L'AUTOMAZIONE D'UFFICIO CHE SI AUTOMODIFICA, Tesi di Laurea in Scienze dell'Informazione, Università degli Studi di Milano, aa. 1984-85

[25] R. Vassallo, APPLICAZIONE DEI SISTEMI ESPERTI ALL'AUTOMAZIONE D'UFFICIO: I COORDINATORI DELLA II GENERAZIONE, Tesi di Laurea in Scienze dell'Informazione. Università degli Studi di Milano, aa 1984-85

[26] M. Heidegger, ESSERE E TEMPO, Longanesi, Milano, 1976

* UNIX è Trademark dei Bell Laboratories

SISTEMI DI INTERAZIONE DIDATTICA

Stefano A. Cerri, Paola Landini

Dipartimento di Scienze dell'Informazione - Università di Milano

Unità di Ricerca in Ingegneria della Conoscenza - Istituto Mario Negri - Milano

RIASSUNTO

La ricerca orientata alla progettazione e realizzazione di sistemi di interazione a scopo didattico ha subito nell'ultimo decennio una profonda evoluzione.
L'utilizzo di tecniche di Intelligenza Artificiale ha portato alla realizzazione di sistemi ICAI (Intelligent Computer Aided Instruction) che differiscono sostanzialmente dai sistemi CAI.
In questo articolo, dopo un esame delle principali differenze, presenteremo i sistemi ICAI da noi realizzati e concluderemo illustrando temi di ricerca e problemi pratici da superare per un utilizzo proficuo di tecnologie informatiche avanzate nell'insegnamento.

ABSTRACT

The research for the design and implementation of interactive systems with didactic purposes has deeply evolved in the last decade.
Artificial Intelligence techniques have been applied for the development of Intelligent Computer Aided Instruction (ICAI) systems which differ substantially from CAI systems.
In this paper we outline the main differences and we present our work in the ICAI research field. We finally address research directions and practical problems for an adequate use of advanced technologies in teaching.

1. APPRENDIMENTO PER IMMERSIONE E PER DIALOGO

I sistemi di interazione didattica possono essere classificati in base alla modalità di interazione prevista con lo studente.
Alcuni esempi sono fondati sulla nozione di apprendimento "per immersione" (o "per scoperta") e forniscono allo studente un ambiente da esplorare per imparare concetti e acquistare capacità senza ricevere una istruzione da parte del sistema.
Un esempio di sistema di questo tipo è LOGO [15] tramite il quale lo studente, programmando i movimenti di una tartaruga per tracciare disegni, impara sia i concetti fondamentali della programmazione che quelli della geometria.
Altri esempi sono costituiti dai giochi matematici come WEST [1] e tutti i sistemi basati sulla simulazione di fenomeni dei quali l'allievo può studiare le proprietà utilizzandone il modello in varie condizioni sperimentali.

Una classe assolutamente diversa, sia come fondamenti concettuali che come struttura, è costituita dai sistemi basati sull'apprendimento "per dialogo" che insegnano in maniera esplicita un particolare campo oggetto.
Fanno parte di questa classe i sistemi di CAI (dall'inglese Computer Aided Instruction) e i sistemi ICAI (Intelligent Computer Aided Instruction).

Se da un lato i sistemi di apprendimento "per immersione" rappresentano per lo studente un ambiente stimolante, è altresì vero che la maggior parte dell'attività didattica è basata sul dialogo docente-studente, e non sembra possibile che questo dialogo possa in tutti i casi essere sostituito da "metafore" stimolanti per l'apprendimento.
Per questo motivo abbiamo scelto di orientare la presentazione che segue all'analisi di sistemi per dialogo didattico, anche se molte delle osservazioni valgono

per i sistemi di interazione didattica in generale.

2. I SISTEMI CAI

I primi sistemi CAI sono stati sviluppati verso la metà degli anni 60.
Tali sistemi si basano su strutture dati ad hoc che vengono messe in anticipo dal progettista del sistema e che contengono il materiale didattico relativo al campo oggetto che il programma intende insegnare.
Di solito essi includono paragrafi di testo, specifiche domande associate alle risposte corrette previste, salti anticipati per un insieme limitato di alternative prevedibili per ogni risposta dell'utente.

I sistemi CAI più sofisticati predispongono percorsi "individualizzati" che si differenziano per qualità, quantità, gradualità di attività e di tempi, modellandosi allo stile di apprendimento di ogni studente (Adaptive Instructional Systems) [13].

Da un punto di vista concettuale-metodologico, la ricerca e lo sviluppo CAI hanno continuato in modo proficuo soltanto in settori ben definiti. Ne ricordiamo due: quello della pianificazione didattica, cioè la scelta delle unità concettuali e la loro sequenzializzazione, e quello della simulazione a scopo didattico di fenomeni matematicamente ben descrivibili.

3. I SISTEMI ICAI

I primi tentativi di applicare tecniche di AI ai sistemi CAI risalgono ai primi anni 70. La considerazione che suggerì di sperimentare tali tecniche è che il modello stimolo-risposta è troppo semplicistico per una interazione didattica efficace [2].
Le ricerche più avanzate posero l'accento sulla necessità di considerare il dialogo didattico nella sua specifica modalità ad iniziativa mista in cui sia il sistema che lo studente possono assumere l'iniziativa di porre domande all'interlocutore.
Queste ricerche, in genere condotte da studiosi di Intelligenza Artificiale o di Psicologia Cognitiva, sono sfociate nella costruzione di sistemi ICAI, più recentemente riferiti anche con la sigla ITS, cioè Intelligent Tutoring Systems[1] (in italiano SII cioè Sistemi Intelligenti per Insegnare).

Quando lo studente assume l'iniziativa, per capire le sue domande e fornire risposte adeguate, il programma didattico deve essere dotato di una interfaccia per la comprensione di un sottoinsieme significativo del linguaggio naturale. Inoltre deve possedere una conoscenza appositamente strutturata che non consista semplicemente di una collezione di domande preconfezionate e di risposte anticipate.

Se la distinzione storica tra sistemi CAI e sistemi ICAI riguarda la capacità di quest'utimi di sostenere un dialogo ad inziativa mista, una descrizione più moderna delle differenze tra le due classi di sistemi è quella che assegna ai sistemi ICAI almeno una delle seguenti caratteristiche:

AUTODESCRIZIONE: modello di se stesso

Il sistema è in grado di fornire informazioni sul proprio comportamento mediante una descrizione del processo di soluzione dei problemi utilizzato eventualmente giustificandolo;

INDIVIDUALIZZAZIONE: modello dello studente

Il sistema è dotato di un modello della conoscenza (statica e comportamentale) dello studente che viene utilizzata per condurre un dialogo per quanto possibile individualizzato cioè orientato alle esigenze dello studente;

CAPACITÀ DIAGNOSTICA: induzione di misconoscenze

il sistema è dotato di strategie diagnostiche, per indurre misconoscenze da errori, che operano scegliendo la descrizione più plausibile del comportamento incorretto dello studente in base a considerazioni cognitive.

Vogliamo sottolineare che lo studio per l'induzione di misconoscenze, a partire da modelli di errore verificabili durante il dialogo diagnostico, distingue la ricerca ICAI all'interno della ricerca sperimentale in Intelligenza Artificiale. Infatti è tipico dell'attività di un docente, umano e automatico, indagare quale conoscenza e quali comportamenti non corretti sono stati costruiti da parte dello studente per risolvere problemi in assenza di informazioni o in presenza di informazioni non corrette.

4. ESPERIENZE DOMESTICHE

L'attività di cui parleremo include sia la ricerca sui sistemi ICAI – in particolare per l'insegnamento di lingue straniere e di linguaggi di programmazione – che la ricerca per la progettazione di sistemi "autore" intelligenti.

DART

[1] Una rassegna sui sistemi ICAI si trova nel libro edito da Sleeman e Brown "Intelligente Tutoring Systems"; Academic Press (1982). Una breve rassegna in italiano si trova in: Cerri S.A. "Sistemi Intelligenti per Insegnare" Nota scientifica S-81-20; Dipartimento di Informatica, Pisa, (1981).

DART è un sistema di programmazione (linguaggio e ambiente) che è stato progettato e sviluppato per facilitare la costruzione di sistemi di interazione didattica da parte di progettisti esperti del campo oggetto ma non necessariamente esperti di Informatica.
Fondamentalmente DART permette una transizione "dolce" da una programmazione tradizionale CAI (nel linguaggio TUTOR) alla sperimentazione con strumenti tipici dell'Intelligenza Artificiale quali Sistemi di Produzioni, Reti di Transizione Aumentate e Reti Semantiche.

Il sistema è stato implementato su PLATO, un sistema dedicato all'istruzione attualmente distribuito dalla Control Data. DART può essere fornito gratuitamente per attività di ricerca.
Una presentazione del sistema si trova in [4], [5].
Il nucleo di DART è stato realizzato anche per un microprocessore MC68000, permettendo di ottimizzare l'architettura del sistema entro limiti accettabili per una stazione di lavoro a singolo utente [6].

ELISA

È un sistema ICAI per l'insegnamento delle lingue che assiste uno studente nella traduzione di parole il cui significato concettuale nella lingua madre dipende dal contesto in cui sono inserite in modo diverso rispetto ad una lingua straniera [4], [7].
ELISA dispone di un semplice modello di comportamenti scorretti che viene utilizzato per indurre, tramite un dialogo diagnostico, la causa degli errori al fine di fornire allo studente dei rimedi adeguati che non siano la semplice constatazione di una risposta incorretta.
In particolare la conoscenza di ELISA consiste di una rappresentazione completa della corrispondenza fra congiunzioni subordinate, contesti (frasi) e concetti identificati dai contesti in italiano e francese. La conoscenza sull'inglese e sull'olandese è limitata, ma sufficiente per esperimenti prototipo.
ELISA è funzionante in DART sul sistema PLATO.

ALICE

È un sistema sviluppato in MRS e Franz Lisp sul Vax 780, UNIX BSD 4.2, per la cui progettazione sono stati affrontati e risolti due problemi identificati durante la ricerca per la realizzazione di ELISA.
Il primo riguarda una classificazione più dettagliata degli errori e delle misconoscenze da essi derivabili per il perfezionamento delle strategie di diagnosi e dei rimedi.
Il secondo consiste nella definizione di una rappresentazione concettuale unica e modulare a cui riferire le congiunzioni di n lingue e da cui dedurre le corrispondenze fra congiunzioni per ogni coppia di lingue incluse nella rappresentazione. Ciò permette ad ALICE di essere un sistema generativo fornendo diversi programmi didattici a seconda della coppia di lingue scelta dallo studente.

Le caratteristiche di modularità della rappresentazione consentono una facile estensione sia a nuove congiunzioni che a nuove lingue [10].
Attualmente ALICE possiede una rappresentazione completa di congiunzioni temporali inglesi, francesi e italiane e di alcune congiunzioni causali delle tre lingue.

TRILL

È un sistema progettato e implementato in MRS e Franz Lisp sul VAX 780, UNIX BSD 4.2. TRILL insegna i fondamenti del LISP ad uno studente che non conosce la programmazione [6], [12].
La strategia diagnostica di TRILL consiste nel risalire la rete semantica di cui dispone proponendo domande e registrando risposte fino a quando lo studente non viene messo in contraddizione con se stesso.
La strategia socratica [18] sembra particolarmente efficace quando è possibile strutturare la conoscenza sotto la forma di catene causa-effetto.

RADAR

È un insegnante intelligente di costrutti iterativi in ADA per studenti che conoscono almeno un altro linguaggio di programmazione [8] (per una versione italiana vedi anche: [9]).

RADAR valuta il codice scritto dallo studente e ne corregge eventuali errori sintattici e semantici costruendo allo stesso tempo ipotesi sull'origine degli stessi in base a preconcetti derivati da nozioni valide in qualche linguaggio di programmazione (tipicamente BASIC, COBOL, FORTRAN e PASCAL) supposto noto allo studente.

5. PROBLEMI

Esistono tre classi di difficoltà che è necessario affrontare per utilizzare proficuamente le tecnologie informatiche avanzate nell'insegnamento.

La prima classe di difficoltà ha a che fare con la *necessità di ricerca fondamentale* nel settore. Soltanto una massa critica significativa di competenze specifiche può garantire di procedere nello sviluppo di applicazioni senza troppi rischi e spesso queste competenze non sono disponibili o non sono facilmente acquisibili.

In particolare *l'acquisizione di conoscenza da esperti* (docenti nel caso particolare) costituice uno dei gravi colli di bottiglia per lo sviluppo di sistemi ICAI.
Esistono due metodi per la soluzione di questo problema.
Il primo consiste nella *acquisizione "mediata"* attraverso un ingegnere della conoscenza che registra i

protocolli di interazione con l'insegnante e li trasforma in modelli di strategie didattiche.
Il secondo metodo consiste nello sviluppare *ambienti di progettazione* di sistemi ICAI (denominati sistemi autore "intelligenti") che permettono all'insegnante stesso di costruire il sistema senza l'intervento dell'ingegnere della conoscenza. È indispensabile che questi ambienti siano dotati di caratteristiche di "amichevolezza", tipiche degli ambienti di ausilio alla progettazione di sistemi in generale, con il vincolo ulteriore che l'utente è quasi certamente non esperto nè di Informatica nè di Intelligenza Artificiale.
Un tema di ricerca molto importante per lo sviluppo di sistemi ICAI efficaci riguarda lo studio di *modelli di comportamento errato* dello studente. Infatti questi sono indispensabili per la costruzione delle strategie di diagnosi che devono essere selezionate all'occorrenza di un errore.
Se da una parte il calcolatore deve "capire" lo studente, anche quest'ultimo deve "capire" l'insegnante automatico. È necessario che i sistemi ICAI siano dotati di *autocoscienza* esplicitabile e comprensibile per lo studente, in modo tale che la giustificazione rappresenti una vera e propria spiegazione nel senso etimologico del termine.
Il sistema ALICE, ad esempio, dispone di modelli di comportamento errato adeguati allo specifico campo oggetto di insegnamento delle congiunzioni, tuttavia non è sempre in grado di spiegare allo studente perchè una particolare congiunzione può oppure no essere usata in un certo contesto.
Una tale spiegazione richiederebbe che la classificazione dei contesti rispetto al significato concettuale della congiunzione presente in essi fosse stata esplicitamente motivata. Poichè non è il caso, ALICE attualmente non è un buon esempio di sistema con "autocoscienza".
Al contrario il sistema RADAR, operando su un campo oggetto formalizzato (costrutti iterativi del linguaggio ADA) è in grado di spiegare allo studente perchè un particolare costrutto iterativo è stato usato incorrettamente.

La seconda classe di difficoltà che ostacola lo sviluppo di applicazioni di sistemi ICAI emerge a causa delle *esigenze di rapporto costi/prestazioni* che questi sistemi devono soddisfare per essere utilizzati in contesti didattici reali.
Bisogna distinguere fra risorse per progettazione e risorse per fruizione di software didattico. In fase di *progettazione,* la classe di sistemi adeguati alla complessità dei problemi identificati sopra coincide con quella per lo sviluppo di Sistemi Esperti [11], con il vincolo aggiuntivo che l'ambiente cui esporre il docente o include l'ingegnere della conoscenza oppure è davvero un sistema autore "intelligente". Questi ultimi ambienti di programmazione non sono ancora disponibili sul mercato e forse non lo saranno a breve termine. La ricerca e lo sviluppo di sistemi come DART tenta di indicare soluzioni a questo problema.
Anche la stazione di lavoro didattica per *fruizione* rappresenta una sfida tecnologica di entità non irrilevante. Infatti, oltre ad essere dotata delle tradizionali caratteristiche delle stazioni grafiche tipiche degli uffici automatizzati avanzati, la stazione didattica deve essere più potente e veloce poichè l'elaborazione per il dialogo didattico è di solito complessa e non si può richiedere ad uno studente di avere la stessa pazienza di un impiegato. Inoltre il costo di fruizione del software deve essere contenuto per essere accessibile da una vasta classe di utenti.

La terza classe di problemi riguarda la *valutazione dei sistemi ICAI.*
Mentre per la valutazione di software tradizionale la componente umana non entra generalmente in modo rilevante come parametro, l'efficacia del software didattico è strettamente legata all'accettazione sia da parte dello studente che del docente. Non si applicano quindi nel caso dei sistemi didattici soltanto i criteri di valutazione tradizionali, come: "il sistema funziona, è robusto, è affidabile ed efficiente". Oltre a questi criteri, è necessario valutare se il sistema si è rivelato pedagogicamente valido.
Di solito quest'ultimo parametro può essere valutato soltanto nell'ambito di gruppi di lavoro che includano competenze psico-pedagogiche adeguate.
Tali competenze devono mirare a progettare strategie di valutazione basate su analisi di protocolli di interazione studente-sistema per un confronto tra apprendimento integrato dal dialogo con il calcolatore e apprendimento in ambiente didattico tradizionale.

6. BIBLIOGRAFIA

[1] Burton R.R., Brown J.S., AN INVESTIGATION OF COMPUTER COACHING FOR INFORMAL LEARNING ACTIVITIES, in "Intelligent Tutoring Systems", Sleeman & Brown (eds), Academic Press, Brighton, 1982, pp. 79-98

[2] Carbonell J.R., AI IN CAI: AN ARTIFICIAL INTELLIGENCE APPROACH TO COMPUTER ASSISTED INSTRUCTION, IEE Trans. Man-Machine Systems MMS-11 (4), 1970, pp. 190-202

[3] Cerri S.A., SISTEMI INTELLIGENTI PER INSEGNARE, Atti Congresso AICA, Pavia, 1981; anche: Nota Scientifica ISI S-81-20, Dipartimento di Informatica, Università di Pisa, 1981

[4] Cerri S.A., Breuker J., A RATHER INTELLIGENT LANGUAGE TEACHER, Proc. AISB Conf. on Artificial Intelligence, Amsterdam, 1980

[5] Cerri S.A., Breuker J., INTELLIGENT DIALOGUES ON PLATO, in: Research in Natural Language Processing in Italy (1981), Cappelli (ed), Giardini, Pisa, 1982, pp. 111-127

[6] Cerri S.A., Fabbrizzi M., Marsili G., THE RATHER INTELLIGENT LITTLE LISPER, Proc. AISB Conf. on Artificial Intelligence and Education, Exeter (UK), 1983
Pubblicato anche in AISB Quarterly 50, 1984, pp. 21-24

[7] Cerri S.A., Merger M.F., LEARNING TRANSLATION SKILLS WITH A KNOWLEDGE-BASED TUTOR: FRENCH-ITALIAN CONJUNCTIONS IN CONTEXT, Proc. 1st European Conference of the ACL, Walker(ed), SRI International, Menlo Park, CA, USA 1983, pp. 133-138

[8] Cerri S.A., Colombini C., Grillo M., Mallozzi R., RADAR: REASONING ON ADA RUBBISH, Proc. 6th European Conference on Artificial Intelligence, Pisa, O'Shea (ed), North-Holland, pp. 409-418

[9] Cerri S.A., Grillo M., Mallozzi M., RAGIONANDO SUGLI ERRORI DI PROGRAMMAZIONE IN ADA, in: Sistemi e Automazione, n. 260, Giugno 1985, pp. 629-636

[10] Cerri S.A., Landini P., MISCONCEPTIONS IN MULTILINGUAL TRANSLATIONS, Proc. 2nd AISB Conf. on Artificial Intelligence and Education, Exeter (UK), 1985

[11] Cerri S.A., PROGRAMS AS TRANSLATIONS: THE REQUIREMENTS OF CONCEPTUAL MODELLING SYSTEMS, Proc. ICAI Research Workshop, Windermere, Cumbria (UK), Aprile 1986, (in stampa)

[12] Daolio M., DIALOGO SUL LISP, Tesi di Laurea in Scienze dell'Informazione, Università di Pisa, 1985

[13] Martinengo G., COME PRODURRE E VALUTARE UN COURSEWARE, SE Scienza Esperienza, Novembre 1985, pp. 25-26

[14] O'Shea T., Self J., LEARNING AND TEACHING WITH COMPUTERS: ARTIFICIAL INTELLIGENCE IN EDUCATION, The Harvester Press, Brighton, 1983

[15] Papert S., MINDSTORM: CHILDREN, COMPUTERS AND POWERFUL IDEAS, Basic Books, New York, 1980

[16] Rovillo S., COMPILAZIONE DI CONOSCENZA: SISTEMI DI PRODUZIONI E RETI DI TRANSIZIONE AUMENTATE SU UN MICROPROCESSORE, Tesi di Laurea in Scienze dell'Informazione, Università di Pisa, 1983

[17] Sleeman D., Brown J.S. (eds), INTELLIGENT TUTORING SYSTEMS, Academic Press, 1982

[18] Stevens A.L., Collins A., THE GOAL STRUCTURE OF A SOCRATIC TUTOR, BBN Rep. * 3518, Cambridge, Mass, 1977

Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE
Via G. Colombo, 49
20133 Milano - Tel. 2772234